THESE DE DOCTORAT

pour obtenir le titre de Docteur en Sciences de l'Université de Nice-Sophia Antipolis

Ecole doctorale « Science et Technologie de l'Information
et de la Communication » de Nice Sophia-Antipolis.

Discipline Informatique

UNIVERSITE DE NICE SOPHIA-ANTIPOLIS
FACULTE DES SCIENCES

JADAPT : UN MODELE POUR AMELIORER LA REUTILISATION DES PREOCCUPATIONS DANS LE PARADIGME OBJET

---

présentée et soutenue publiquement par

laurent QUINTIAN

Thèse dirigée par

philippe LAHIRE

Le 13 Juillet 2004 au laboratoire I3S, devant le jury composé de

Président du Jury	OLIVIER LECARME	Professeur, Université de Nice Sophia-Antipolis
Rapporteurs	LAURENCE DUCHIEN	Professeur, Université des Sciences et Techniques de Lille
	GILLES ROUSSEL	Maître de Conférences, HDR, Université de Marne-la-Vallée
Directeur de thèse	PHILIPPE LAHIRE	Maître de Conférences, Université de Nice Sophia-Antipolis

TABLE DES MATIERES

Chapitre 1

Introduction

De nos jours, les applications intègrent des fonctionnalités qui alourdissent le développement comme la distribution, la concurrence, les interfaces homme-machine, les contraintes temps réel, la persistance, et la tolérance aux pannes. Leur implémentation peut être soumise à des changements qui ne sont pas forcément prévus dès l'origine et leur interdépendance n'est donc pas nécessairement prise en compte. Les applications (et leur implémentation) deviennent donc de plus en complexe et difficile à faire évoluer. Les fonctionnalités précédemment citées peuvent aussi être partagées entre plusieurs applications. Or leur réutilisation est rendue difficile à cause des interdépendances. Un des buts du génie logiciel est de faire face à ces problèmes ; les paradigmes de programmation facilitent l'implémentation des applications. Par exemple, la programmation orientée objets qui a succédé à la programmation procédurale, apporte une meilleure encapsulation et une meilleure décomposition en unissant les données et les traitements. Pourtant, la programmation orientée objets ne prend pas en compte les fonctionnalités dont l'implémentation est transversale à la hiérarchie de classes. L'encapsulation constitue donc un frein à la réutilisation et à l'évolution des fonctionnalités. Des méthodologies récentes comme la conception par *framework* n'arrivent pas non plus à « capturer » ces fonctionnalités de manière efficace. Différents constats montrent que les approches qui viennent d'être évoquées sont loin de répondre aux besoins des applications.

La séparation des préoccupations [LH 95] est une suite aux articles de [Par 72] et [Dij 76] qui propose un nouveau paradigme qui se place au-dessus du paradigme de l'objet pour alléger la hiérarchie des classes des diverses fonctionnalités. Dans ce paradigme on utilise le terme de *préoccupation* pour désigner les diverses fonctionnalités d'une application comme celles mentionnées au début de l'introduction. Ce paradigme identifie les préoccupations qui composent une application puis les sépare lors de la phase de conception de l'application. Cette séparation doit ensuite être répercutée dans l'implémentation de l'application. Le paradigme objet permet d'isoler toutes les classes d'une même préoccupation dans un espace de nom. Si l'on prend garde à éviter le couplage entre ces espaces de nom, le paradigme de l'objet permet de séparer les préoccupations. Cette séparation fait ensuite intervenir une phase de composition (le plus souvent avant la phase de compilation) des préoccupations afin de constituer le code source de l'application. La composition nécessite donc une description des relations de dépendance (couplage) entre les préoccupations de l'application. Les relations de dépendance peuvent être décrites par des *opérateurs d'adaptation*. Une description des adaptations à effectuer permet donc aux langages (dont la plupart ne sont que des extensions de langages existants) qui supportent la séparation des préoccupation de composer les différentes préoccupations pour former l'application finale. La délocalisation du couplage dans la description des adaptations est la propriété principale du paradigme de séparation des préoccupations ; c'est cette propriété qui améliore la réutilisation et l'évolution des applications. Néanmoins, cette délocalisation est aussi la source de la principale difficulté rencontrée : la composition. Elle se traduit par une difficulté à décrire les adaptations nécessaires à la composition de l'application. Pour simplifier ce problème, la plupart des approches actuelles utilisent le fait qu'une partie des adaptations nécessaires à la réutilisation d'une préoccupation est indépendante du contexte d'utilisation et donc commune à toutes les réutilisations de la préoccupation. Les adaptations nécessaires à la composition d'une préoccupation peuvent donc s'abstraire et se généraliser, nous appelons cette abstraction le *protocole de composition* d'une préoccupation. Ce protocole doit être spécialisé pour réutiliser la préoccupation dans le contexte d'une application don-

née. Les protocoles de composition permettent de simplifier la composition nécessaire à la séparation des préoccupations, il n'est plus nécessaire d'écrire toute la composition d'une préoccupation mais seulement de spécialiser son protocole de composition. Le protocole de composition sert aussi de garantie minimale à la composition. Pourtant, nous allons voir dans l'état de l'art qu'aucun langage ou modèle ne permet d'encapsuler complètement les protocoles de composition. De plus, la spécialisation n'est pas supportée par toutes les approches. L'état de l'art ne fournit donc pas une aide complète lors de la composition ; cette étape est pourtant la difficulté principale du paradigme de séparation des préoccupations.

Le travail présenté dans cette thèse propose une solution pour faciliter la réutilisation des préoccupations. Pour y parvenir nous allons utiliser au maximum le potentiel des protocoles de composition. Nous avons fait une étude de l'état de l'art pour cerner les besoins nécessaires à l'encapsulation et à la spécialisation des protocoles de composition. Cette étude utilise trois exemples suffisamment variés qui montre les avantages et les limites des approches existantes. Elle nous a permis de dresser un cahier des charges pour la réutilisation des préoccupations. Ce cahier des charges a ensuite permis de construire un nouveau modèle. Ce modèle encapsule les protocoles de composition et leur spécialisation par une même entité que nous nommons l'*adaptateur*. L'adaptateur est une entité de première classe qui contient des adaptations qui ne sont d'ailleurs pas toutes supportées par les approches existantes. Elle peut s'abstraire et encapsule alors un protocole de composition. Elle peut aussi être spécialisée par héritage pour réutiliser une préoccupation dans un contexte donné. Pour valider le modèle, nous l'avons implémenté avant de le confronter aux exemples présentés dans l'état de l'art.

Cette thèse cible les langages à classes fortement typées (comme C++, Java [JAV 04], C#, Eiffel), pour lesquels des outils pour faire de la séparation des préoccupations existent. Le chapitre 2 présente l'état de l'art de la séparation des préoccupations dans le paradigme objet et sélectionne un ensemble de modèle et de langage. Le chapitre 3 étudie leurs apports et limitations à travers la mise en œuvre de plusieurs exemples concrets. Il dresse ensuite un bilan des besoins de la séparation des préoccupations. Celui-ci donne lieu à l'élaboration d'un cahier des charges dans le chapitre 4. Son principal objectif est de faciliter la réutilisation des préoccupations. Le cahier des charges conduit dans le chapitre 5 à la description du modèle proposé pour réaliser la séparation des préoccupations. Le chapitre 6 décrit une implémentation (nommée JAdapt) d'un *plug-in* pour l'environnement de développement Eclipse [ECL 04] qui permet d'utiliser notre modèle pour le langage Java. Le chapitre 7 évalue notre implémentation avec les exemples du chapitre 3 et propose une comparaison de JAdapt avec les approches existantes. Nous présentons finalement trois perspectives dans le chapitre 8 : l'amélioration de notre implémentation, la réutilisation de notre modèle dans le paradigme des composants et enfin la réutilisation de notre modèle dans le paradigme MDA [MDA 04].

Chapitre 2

La séparation des préoccupations : un état de l'art

La complexité grandissante des systèmes actuels met en évidence certaines limites du paradigme de l'objet. La séparation des préoccupations est une approche qui apparaît comme prometteuse pour répondre à ces limites. Ce chapitre est un état de l'art de différentes approches offrant un support à la séparation des préoccupations dans le paradigme de l'objet. Cette étude, tout comme cette thèse, se restreignent aux langages à classes dans un environnement compilé et fortement typé. L'objectif de cette étude est de déterminer un ensemble d'approches dont l'étude doit être approfondie. Une étude approfondie sera ensuite réalisée sur ces différentes approches pour mettre en évidence l'ensemble des critères qui nous semblent essentiels[1] et qui serviront de support pour le modèle que nous proposons.

Cet état de l'art se décompose en plusieurs sections. La section 2.1 propose une brève introduction au paradigme de l'objet qui se focalise sur ses capacités de modularisation. La section 2.2 étudie le paradigme de la séparation des préoccupations. La section 2.3 expose des différents modèles généraux qui offrent un support pour la séparation des préoccupations. La section 2.4 fait un inventaire des différents langages qui reposent sur plusieurs modèles précédemment cités. A partir de la réflexion menée dans les deux sections précédentes, la section 2.5 retient plusieurs approches qui semblent s'adapter plus particulièrement à notre problématique. Le Chapitre 3 approfondit l'étude de ces approches.

2.1 APPORTS ET LIMITITATIONS DU PARADIGME DE L'OBJET

La programmation orientée objets prend ses racines dans l'approche modulaire de langages comme Simula [BDM 73], Pascal [KN 74] ou Modula qui s'appuient sur plusieurs concepts clés. Les *objets* réifient des données et les opérations qui leurs sont associées ; c'est le principe d'*encapsulation*. Un objet encapsule un état (ses variables) et les opérations possibles sur cet état (ses *méthodes*).

L'encapsulation permet au monde extérieur d'accéder à un sous-ensemble de ces méthodes (celles qui sont déclarées *publiques*). L'état d'un objet n'est accessible qu'à travers ses méthodes ; cette propriété permet à un objet de toujours garder un état cohérent. Un objet est souvent comparé à une boite noire, qui n'est activable qu'au moyen de l'*envoi de message* [GR 83] et [Duc 99] (un message correspond, en général, à une méthode).

Un langage à objets est dit réflexif s'il dispose d'une représentation manipulable (réification) des diverses entités qui forment un programme comme les classes ou les méthodes, et qui permettent leur exécution (ex : pile d'exécution, mémoire, etc.). Selon les langages, ces diverses entités peuvent être observées (*introspection*) et/ou modifiées (*intercession*).

Chaque objet possède un type qui est décrit par une classe[2]. On dit de lui qu'il est une *instance* de son type. La classe décrit la *structure de* ses instances (variables) ainsi que leurs comportements (méthodes). Elle fabrique les instances au moyen d'une méthode spéciale appelée *constructeur*.

[1] Notamment pour la prise en compte de la séparation des préoccupations.
[2] Une classe, selon qu'elle est générique ou pas, correspond à un ou plusieurs types.

Les classes comme leurs instances ont un état (il est représenté par l'ensemble des variables de classe) ainsi que des méthodes (dénommées méthodes de classe) qui permettent d'agir sur cet état.

Toutes les instances d'une même classe s'appuient sur les mêmes variables et les mêmes méthodes. Néanmoins, les valeurs associées aux variables sont propres à chaque instance. Une propriété intéressante est que les instances d'une même classe sont donc toutes isomorphes.

Enfin, les classes peuvent posséder une relation d'*héritage* entre elles. Une classe hérite d'une autre (ou de plusieurs autres classes dans le cas de l'*héritage multiple*). La classe « fille » est vue comme une extension de la classe « mère » ; elle hérite de la structure de sa classe mère. Une classe fille est compatible à la fois sur les plans structurels et comportementaux avec tous ses ancêtres[1]. C'est le principe de *compatibilité ascendante* entre les classes. Les instances bénéficient de ce principe : une instance est conforme avec toutes les classes ancêtres de sa classe.

Pour conclure, le paradigme objet permet d'implémenter des applications de manière modulaire (grâce à l'encapsulation), de construire des ensembles d'objets d'une même classe (lien d'instanciation), d'améliorer incrémentalement leurs possibilités (héritage) tout en conservant une compatibilité ascendante (sous-typage).

Ces caractéristiques font du paradigme de l'objet une approche qui favorise la réutilisation et l'évolution du code. Pourtant, la modularité offerte par le paradigme de l'objet peut poser certains problèmes [OM 01] que nous allons détailler dans le chapitre 3.

2.2 QU'EST-CE QUE LA SEPARATION DES PREOCCUPATIONS ?

Les applications modernes ont besoin de plus en plus de fonctionnalités afin de faire face à des problèmes toujours plus nombreux ; on peut citer par exemple : la distribution et la concurrence des données, les interfaces homme-machine, la prise en compte du temps réel, la gestion de la persistance ou la robustesse aux pannes. L'introduction de ces fonctionnalités alourdit le développement pour différentes raisons : *i)* leur implémentation est tributaire de l'évolution des technologies (plateformes logicielles), *ii)* elles sont rarement prises en compte et identifiées lors de la phase de conception.

La présence de ces diverses préoccupations pénalise les capacités d'évolution de l'application. En effet, leur implémentation dans le paradigme objet est souvent transversale à la hiérarchie de classes, et a tendance à la « polluer ». Même l'utilisation (au dessus des concepts objets) de méthodologies évoluées, comme la conception par *framework* [FSJ 99], n'arrivent pas à « capturer » ces préoccupations de manière efficace. Les constructions syntaxiques offertes par les langages associés au paradigme objet ne sont pas suffisantes. En particulier, le fait d'avoir le concept de classe comme unique entité d'encapsulation ne permet pas d'inclure les nombreuses préoccupations des applications de manière séparée.

Pour satisfaire l'intégration de nouvelles préoccupations, on a vu apparaître de nombreuses extensions de langages largement diffusés comme C, C++, Java, ou Eiffel. Ces langages sont dédiés à un ensemble de préoccupations figé (distribution et concurrence, temps réel, etc.) et ne résolvent que des sous-ensembles des besoins des applications.

Ce besoin des applications a donné naissance à un nouveau paradigme : « la séparation des préoccupations »[2] [LH 95]. Ce paradigme prône la séparation des préoccupations à la fois pendant la conception [CG 99] et pendant l'implémentation.

---

[1] Cependant certains langages (C++ par exemple) peuvent étendre la sémantique de l'héritage, en lui adjoignant des modificateurs de visibilité (comme pour les méthodes des objets).

[2] Le terme anglais est « Separation of Concerns ».

La séparation des préoccupations met en évidence trois tendances : *i)* les applications sont composées de différentes préoccupations, *ii)* la complexité des applications est en constante augmentation, et *iii)* le nombre de préoccupations requis par les applications est lui aussi en augmentation.

La séparation des préoccupations a provoqué l'émergence de nouveaux langages « généraux » de programmation (dont la plupart ne sont que des extensions de langages existants), capables de séparer les préoccupations pendant l'implémentation. En parallèle, on constate l'apparition de méthodologies de développement [Nak 00] et de *reengineering* [Ken 00].

La séparation des préoccupations propose d'identifier les différentes facettes d'un système comme : les parties fonctionnelles (structures et comportements qui correspondent à la partie métier de l'application) et les parties non fonctionnelles (code de synchronisation, affichage de traces, traitement de la persistance, gestion de transaction, etc.). Cette première distinction entre les différentes catégories de préoccupations, va dans le sens d'une conception et d'une implémentation simplifiée, d'une meilleure compréhension, d'une diminution du couplage entre préoccupations et plus généralement, d'une réutilisation accrue.

Il est important que la séparation des différentes préoccupations soit réalisée à la fois au niveau conceptuel et au niveau de l'implémentation. Comme cela va être montré par la suite (voir chapitre 3) le paradigme objet ne permet pas de satisfaire ces hypothèses. Car l'implémentation des préoccupations se trouve éparpillée dans un ensemble de classes ou composées à l'intérieur d'une même méthode. En effet, la modularité naturelle de certaines préoccupations s'entrecroise avec le reste de l'implémentation. De ce fait le paradigme objet augmente le couplage entre les différentes entités de l'application.

Une préoccupation est un concept générique décrivant une entité homogène composant le logiciel. La notion de préoccupation induit de nouvelles questions lors de la conception et de l'implémentation : quelles sont les entités qui ont vocation à devenir une préoccupation ? Ou bien, de quoi une préoccupation est-elle formée ? On revient au problème d'identification des différentes sortes de préoccupation.

Les préoccupations doivent être séparées les unes des autres pour permettre leur réutilisation dans différents contextes. La difficulté est de s'affranchir des problèmes de couplage inter-préoccupations.

Par rapport au paradigme de l'objet, celui de la séparation des préoccupations permet de gagner en modularité et en expressivité que ce soit pour la conception ou l'implémentation. Pour être réutilisées, les préoccupations sont composées entre elles et le reste de l'implémentation de l'application. Cette phase de composition nécessite le plus souvent des adaptations mutuelles entre les différentes préoccupations. La séparation des préoccupations délocalise donc un ensemble de problèmes liés à la réutilisation vers la phase de composition.

Ce « tissage » des différentes préoccupations implique une composition le plus souvent non orthogonale[1], c'est-à-dire que des préoccupations peuvent modifier la sémantique d'autres préoccupations. Elle s'oppose à une composition de préoccupations orthogonales où l'intersection de la sémantique des différentes préoccupations est nulle (ce qui revient à dire que les préoccupations sont toutes indépendantes les unes des autres).

La séparation des préoccupations a été améliorée par la suite. Cette amélioration est connue sous la dénomination de séparation multidimensionnelle des préoccupations. Elle correspond à une séparation des préoccupations qui satisfait aux pré-requis suivants :

[1] Une composition non orthogonale implique que les différents éléments à composer ne peuvent l'être de manière séquentielle. Pour être composés les comportements doivent être tissés les uns avec les autres.

– Pouvoir classer les préoccupations dans plusieurs dimensions arbitraires ; une dimension représente un ensemble de préoccupations ayant des caractéristiques communes.

– Une séparation des préoccupations simultanée sur un nombre arbitraire de dimensions, sans pour autant avoir une dimension privilégiée qui entraînerait la décomposition le long des autres dimensions d'une manière favorisée.

– Pouvoir identifier et encapsuler n'importe quel type de préoccupation.

– Etre capable de supporter de nouvelles préoccupations et de nouvelles dimensions de préoccupation lors de leur introduction pendant le cycle de développement.

– Prendre en compte les préoccupations qui présentent entre-elles un couplage et dont le comportement peut donc interférer avec celui d'une ou plusieurs préoccupations[1].

Une implémentation qui offrirait un support complet de la séparation multidimensionnelle des préoccupations permettrait de fait, une *remodularisation à la carte*. Pour un développeur, cela aurait pour principal intérêt de pouvoir choisir l'unité de modularisation la plus adéquate, en fonction d'une ou plusieurs préoccupations.

La séparation multidimensionnelle des préoccupations représente un ensemble d'objectifs très ambitieux. Ils s'appliquent aussi bien aux langages qu'à la conception d'application. Néanmoins, aucune technique existante ne satisfait ces objectifs et une importante activité de recherche doit encore être menée [THO 00].

Dans la suite de ce document nous utiliserons pour plus de simplicité l'expression « séparation des préoccupations » en lieu et place de « séparation multidimensionnelle des préoccupations ».

2.3 MODELES SUPPORTANT UNE SEPARATION DES PREOCCUPATIONS

Cette section présente les différents modèles supportant (avec un degré variable) la séparation des préoccupations. Il est à noter que nous n'avons retenu que les modèles utilisables dans le paradigme de l'objet.

2.3.1 Programmation générique

Un langage qui supporte la programmation générique permet de paramétrer les classes. Une classe ainsi paramétrée est dite générique ; elle possède un ou plusieurs paramètres génériques formels qui définissent son degré d'ouverture. Ceci permet, par exemple, de paramétrer une classe représentant une structure de données (comme une collection) avec le type des instances qu'elle contient. La programmation générique permet donc de séparer (découpler) la préoccupation liée à la définition d'une structure de données de la spécification du contenu qu'elle permet de mémoriser et de manipuler. La composition est effectuée lors de l'instanciation de la classe générique (ce sont les paramètres génériques effectifs qui fournissent l'information complémentaire qui est nécessaire à l'instanciation).

De nombreux langages de programmation à objets supportent la programmation générique : CLOS, C++, Eiffel, Java (à partir de la version 1.5), etc.

Les *Mix-In* [BC 90] et [CE 99] et les *Mix-In Layers* [SB 98] sont deux techniques dérivées de la programmation générique qui améliorent la séparation des préoccupations.

Un mix-in est une classe générique dont le paramètre générique est sa super-classe. Un mix-in est une entité qui permet d'implémenter une extension qui est orthogonale à un ensemble de classes.

[1] Cette propriété est importante car l'expérience montre que des préoccupations à priori orthogonales le sont en fait très rarement.

L'approche à base de Mix-In Layers correspond à une conception par couche : chaque couche conceptuelle représente une collaboration entre différents objets de l'application qui peuvent eux-mêmes, jouer des rôles différents selon les collaborations (voir figure 1).

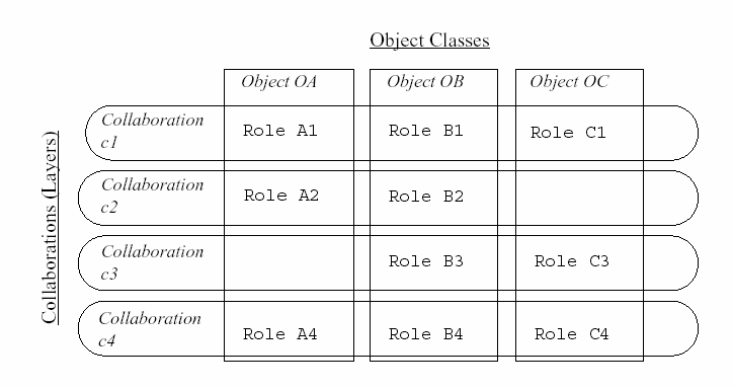


Figure 1. Exemple d'une décomposition en collaborations. Les ovales représentent les *collaborations*, les rectangles représentent les *objets*, les intersections représentent les *rôles*. Cette figure provient de [SMA 98].

Une collaboration est implémentée par un *mix-in* ayant des paramètres génériques supplémentaires pour représenter les rôles. Les Mix-In Layers permettent une double séparation entre les différents rôles joués par les objets et les différentes collaborations. Il est difficile de composer des Mix-In Layers entres eux à cause, en particulier, de l'importance de l'ordre d'instanciation des paramètres.

La programmation générique ne sera pas étudiée de manière plus détaillée dans le chapitre 3 car elle correspond à une approche trop restrictive de séparation des préoccupations. Néanmoins, les solutions proposées dans le chapitre 3 prennent en compte les apports de la programmation générique pour la séparation des préoccupations.

2.3.2 Métaprogrammation

Nés du besoin d'ouverture des langages, les protocoles à méta objets [KRB 91] formalisent le comportement d'un langage à objets à l'intérieur de *métaobjets* qui implémentent la sémantique du langage.

Ces métaobjets sont associés à une ou plusieurs classes qu'ils décrivent fonctionnellement et structurellement. Il est donc nécessaire de leur associer un méta-langage pour les exprimer. Ce méta-langage peut être vu en première approximation comme une extension d'un langage à objets ; c'est le début de l'unification des métaobjets et des objets. Les protocoles à métaobjets vont utiliser les abstractions fournies par les langages à objets.

Une classe est une abstraction qui exprime le comportement commun d'un ensemble d'objets (ses instances). Une métaclasse est une abstraction qui permet de décrire le comportement d'une ou plusieurs classes (ses instances). Ces différentes entités constituent un graphe de hiérarchie d'instanciation : la métaclasse instancie la classe qui elle-même permet d'instancier des objets. Par contre, les objets ne peuvent instancier de nouvelles entités. Cette hiérarchie d'instanciation a pour effet d'associer à *chaque* classe une métaclasse, cette association est aussi appelée *lien méta*.

De même que pour les classes, une relation d'héritage est définie pour les métaclasses. Il faut une méta classe « racine » qui dans un système réflexif va être sa propre instance et hériter de la

classe Objet pour être vue comme un objet. Ceci nécessite une méta-circularité[1] : un système à métaobjets doit donner une définition récursive (et non infinie) de la première métaclasse (par auto amorçage).

Associer un protocole à métaobjets à un langage permet de l'ouvrir pour le rendre réflexif. La réflexion permet d'étendre et d'adapter le langage aux besoins de l'utilisateur de manière incrémentale et modulaire. Il suffit de modifier les métaobjets pour modifier le langage, tout en restant dans le paradigme objet grâce à l'héritage.

Les métaobjets ont une vocation plus générale que d'autres modèles en ce qui concerne l'adaptabilité, car ils agissent directement au niveau du langage, en proposant d'ouvrir celui-ci aux modifications du développeur. C'est le protocole à métaobjets, qui identifie les points d'accès (appelé points de jointures dans la terminologie de la séparation des préoccupations) du langage pour permettre des modifications comportementales de celui-ci.

Pour les mêmes raisons que la programmation générique nous avons décidé de ne pas étudier les différents méta-modèles disponibles. Toutefois, l'étude du chapitre 3 tiendra compte des apports de la métaprogrammation pour la séparation des préoccupations.

2.3.3 Programmation par rôles ou points de vue

La programmation par rôles [GS 96] ou par points de vue [BC 96] et [AB 91] sont deux paradigmes assez proches ; ils permettent de déstructurer la notion d'objets par l'introduction de vues subjectives : les rôles qu'ils jouent ou les points de vue qu'ils représentent. Les rôles ou les vues sont donc les préoccupations dont ces deux modèles formalisent la séparation.

Même, si conceptuellement, ces deux paradigmes permettent de supporter la séparation des préoccupations, leurs différentes implémentations ne permettent pas de réconcilier les problèmes de partage d'information et de recouvrement entre les différentes préoccupations. Notons, néanmoins, qu'un langage étudié dans la section suivante (Jiazzi [MH 03]) permet de programmer par rôles ou par points de vue.

2.3.4 Programmation par aspects

La programmation par aspects [KLM 97] est un modèle de séparation des préoccupations basé sur le paradigme objet (bien que le modèle ait été par la suite appliqué à d'autres paradigmes). Pour un état de l'art de la programmation par aspects se référer à [BL 01]. Ce modèle formalise dans les applications l'entrelacement des préoccupations qui sont mal prises en compte par le paradigme objet. Cet entrelacement (appelé aussi *crosscutting*) est le résultat d'une opération de tissage entre les différents fils que l'on assimile à des préoccupations. Il résulte de l'absence de séparation entre les préoccupations composant une application. La programmation par aspects propose de nouvelles entités pour remédier à cet entrelacement.

Les éléments qui s'entrelacent sont nommés *aspects*. Un aspect est une abstraction d'une préoccupation, dont la particularité est de s'appliquer à un ensemble de classes. L'implémentation d'un aspect va s'entrelacer avec le reste de l'implémentation. Le paradigme objet est par contre toujours utilisé pour modéliser les comportements qui sont naturellement hiérarchisés.

Enfin, comme les aspects sont des modules à part entière, il faut pouvoir les composer avec le reste du système. Les endroits où les aspects interviennent avec le reste du programme sont appelés *points de jointure*[2].

---

[1] Pour une solution élégante voir l'approche proposée par ObjVlisp [Coi 87].

[2] Les points de jointures sont des éléments de la sémantique du langage sur lesquels les aspects peuvent agir par adaptation.

Un point de jointure peut avoir une granularité variée : une méthode particulière, un ensemble de méthodes, toutes les méthodes d'une classe, toutes les méthodes d'un ensemble de classes, etc. Chaque point de jointure possède une information contextuelle associée qui est utilisable par l'aspect pour savoir où ce dernier s'applique.

Les aspects possèdent des méthodes[1] qui sont attachées à un ou plusieurs points de jointures. Une fois attachée à un point de jointures, la méthode est exécutée chaque fois que le flot d'exécution du programme atteint ce point. Un modificateur peut préciser le moment d'exécution par rapport au point de jointures : avant, après, autour, après exception ou encore, après retour de valeur. De plus, ces méthodes possèdent une variable d'instance supplémentaire nommée *thisJoinPoint* qui encapsule l'information contextuelle capturée depuis la jonction en cours : message intercepté, classe adressée par le message, paramètres, etc.

Une étape supplémentaire doit être ajoutée pour exécuter un programme avec des aspects : soit une nouvelle étape de compilation qui va composer les aspects et le programme « normal » avant de l'envoyer au compilateur natif du langage, soit une modification de l'interprète pour pouvoir composer les aspects à l'exécution. Cette étape est nommée « tissage des aspects » (*aspect weaver*) ; elle sous-entend ainsi qu'à un moment donné l'exécution des aspects et des objets sont composées pour fournir le comportement escompté. Pour des exemples d'implémentation de tisseur d'aspect, voir [FS 98] ou [DLB 01].

Les aspects possèdent la capacité d'hériter d'autres aspects et d'encapsuler des variables et des méthodes d'instance ou de classe. Ils peuvent donc être instanciés et plusieurs politiques d'instanciation sont envisageables : une instance d'un même aspect pour tous les objets d'une classe, chaque objet peut posséder une instance propre d'un même aspect, ou encore une seule instance commune.

Dans le cas d'un environnement réflexif, les spécificités des aspects doivent aussi répondre au besoin de réflexivité. Il faut alors que l'implémentation des aspects soit complètement dynamique, pour un exemple voir [SMM 03]. Pour permettre, par exemple, de changer un aspect à l'exécution.

En conclusion, la programmation par aspects permet d'implémenter des préoccupations qui s'entrelacent avec le reste du système d'une manière modulaire. Ce paradigme permet de bénéficier des avantages de la modularité qui garantie un code plus simple, plus facile à développer et à maintenir et qui a un meilleur potentiel de réutilisation. La section 4 présente différents langages de programmation par aspects.

2.3.5 Programmation par sujets

La programmation par sujets [HO 93] est (comme la programmation par aspects) une extension du paradigme de l'objet. Un *sujet* est un programme ou un fragment de programme exprimé dans le paradigme objet. La programmation par sujets propose de composer un ensemble de sujets pour produire l'application finale. Ce processus de composition est nommé *intégration*.

Les sujets sont tous vus au même niveau, aucun sujet n'est *a priori* dominant par rapport à un autre. Les sujets peuvent être composés entre eux pour produire des sujets plus importants, qui peuvent être à leur tour composés entre eux.

L'intégration des sujets se fait en définissant des règles [OKHKK 95]. Une règle implique au moins deux sujets, elle permet de définir par des opérations simples (fusion, redéfinition et séquencement) la composition.

L'intégration est encapsulée par des modules de composition qui mettent en relation plusieurs sujets à intégrer ensemble.

---

[1]Les méthodes qui peuvent s'attacher à des points de jointures sont nommées *advice*.

Le paradigme de la programmation par sujets possède un certain nombre d'avantages pour la conception et l'implémentation :

– Possibilité d'extension et de composition non prévue *à priori*, sans changer ou recompiler le code source existant. Pour cela, il suffit d'encapsuler dans un sujet les extensions et de fournir les règles de composition pour l'intégrer avec les autres.

– Développement de classes décentralisées [OHBS 94]. Les auteurs de différentes applications qui partagent les mêmes objets peuvent avoir leurs propres définitions de vues sur les objets partagés. Ces définitions peuvent ensuite être composées.

– Développement indépendant des préoccupations. L'implémentation des comportements peut être construit comme un sujet cohérent, plutôt que d'être entrelacé avec le reste du code.

De plus, ce paradigme a évolué pour permettre de diminuer encore le couplage entre les sujets, permettant ainsi une meilleure séparation des préoccupations ; cette évolution s'appelle la programmation par *HyperSpace*.

La programmation par *HyperSpace* [OT 00] [HYP 03] est à la fois une instance de la séparation multidimensionnelle des comportements et une extension de la programmation par sujets. Sa principale particularité est d'imposer la propriété de complétude aux sujets[1]. Une préoccupation est encapsulée dans un *HyperSlice* et les règles de composition sont décrites dans des *HyperModules*. Enfin, une phase de composition nommée *intégration* réunit tous les comportements pour former l'application finale.

Les trois activités liées à la séparation des comportements : identification, encapsulation et intégration sont prises en charge par le modèle des *HyperSpaces*.

– L'identification est la sélection d'une préoccupation, elle contient les classes et les méthodes qui lui sont pertinentes. La préoccupation est alors représentée par un *HyperSpace*.

– Les préoccupations sont ensuite encapsulées pour être manipulées comme des classes, dans différents *HyperSlices*. Un *HyperSlice* peut contenir une ou plusieurs préoccupations.

– Finalement, pour intégrer correctement les différentes préoccupations, les *HyperModules* définissent les règles de composition.

Un *HyperSlice* comporte un ensemble de classes (ou même de procédures) qui doivent être complètement déclarées : il ne peut *référencer* des objets (des méthodes, des classes, etc.) qui lui sont extérieurs. Toutefois un *HyperSlice* ne requiert pas une définition complète de ces déclarations : par exemple une fonction peut être spécifiée sans pour autant être implémentée. Cette qualité de *complétude* de *l'HyperSlice* permet d'éliminer le couplage entre les différents *HyperSlices*. Les *morceaux manquants* sont obtenus lors de la phase d'intégration. En d'autres termes, les *HyperSlices* sont complétés dans les *HyperModules*.

C'est une propriété des *HyperSlices* d'avoir une sorte de relation polymorphique entre différents *HyperSlices*. Ainsi, plusieurs *HyperSlices* peuvent représenter la même préoccupation pour, par exemple, faciliter les tests. Les *HyperSlices* sont, en quelque sorte, des briques de construction, qui peuvent être intégrées pour former des briques de construction plus grandes ou un système complet.

Un *HyperModule* comprend à la fois un ensemble d'*HyperSlices* et des relations d'intégration qui spécifient la relation entre ces *HyperSlices* et la manière de les intégrer. Un *HyperModule* peut servir d'*HyperSlice* pour un autre *HyperModule* ; on peut donc le considérer comme un *HyperSlice* composite.

[1] Voir plus bas la définition d'*HyperSlice*.

Par la suite, nous emploierons respectivement, le terme de « programmation par sujets » pour « programmation par *HyperSpace* », celui de « sujet » pour « *HyperSlice* », et celui de « module de composition » pour « *HyperModule* ».

On peut dire en conclusion que la programmation par sujets offre un support intéressant pour la séparation des préoccupations. La section 4 présente différents langages de programmation par sujets.

2.3.6 Filtre de composition

Les filtres de composition [ABV 92] ajoutent aux objets des filtres qui interceptent l'envoi et la réception de messages. Les messages entrants (respectivement sortants) sont soumis à l'ensemble des filtres d'entrée (respectivement de sortie). Cette fonctionnalité est obtenue à travers la réification des envois de message.

Un filtre est lui-même un objet qui évalue et manipule des messages réifiés. Un filtre peut accepter ou rejeter un message tout en déclenchant un certain nombre d'actions. Les filtres sont ordonnés dans un ensemble (de filtres d'entrée ou de sortie), c'est leur seule capacité de composition immédiate. Les filtres sont tous orthogonaux deux à deux.

Un message n'est reçu par l'objet destinataire qu'après avoir passé tous les filtres d'entrée. De même, un message ne quitte l'objet émetteur qu'après avoir passé l'ensemble des filtres de sortie. Les filtres sont donc liés par un opérateur booléen « et ».

Chaque objet peut avoir des conditions, qui reflètent un état possible de l'objet. Ces conditions sont définies dans les objets, sous forme de méthodes sans argument renvoyant une valeur booléenne.

A chaque filtre correspond une expression de filtrage. Cette expression doit être définie en respectant la forme générale suivante :

```
(Condition Operator Pattern Parameters)+
```

La `condition` est une expression booléenne obtenue par composition des diverses méthodes booléennes exhibées par l'objet.

L'`operator` spécifie l'effet que doit produire le filtre si la *condition* est vraie. Deux opérateurs sont définis dans le modèle : l'*inclusion* et l'*exclusion*. L'inclusion provoque l'acceptation de tous les messages qui correspondent au `pattern`[1]. L'exclusion provoque l'acceptation de tous les messages qui ne correspondent pas au `pattern`. Les `paramètres` sont optionnels et dépendent fortement du type de filtre auquel cette expression est associée.

Les filtres s'intègrent relativement bien dans le paradigme objet [Ber 94]. Chaque objet peut avoir un ensemble de filtres d'entrée et de sortie. Les filtres sont réifiés par des objets. Une expression de filtrage peut être vue comme un arbre lui aussi réifié par un objet.

Lorsque les langages à objets au-dessus desquels les filtres de composition sont ajoutés sont réflexifs, alors les filtres doivent aussi offrir une capacité de réflexion.

Cependant, les filtres de compositions ne se conforment pas totalement à la philosophie du paradigme objet car les ensembles de filtres en entrée et en sortie ne peuvent pas être hérités. Le modèle (ainsi que les diverses implémentations) ne propose aucune manière de redéfinir ou d'utiliser des filtres d'une classe héritée. C'est une lacune assez importante qui limite la réutilisation et le partage des ensembles de filtres.

---

[1] La forme générale est : `[nomCible | * ].[nomMethode | *]`.

Les filtres, par leur impact sur les messages, modifient le comportement des objets sur lesquels ils sont appliqués. Cette fonctionnalité permet de remonter certaines préoccupations associées aux objets au niveau des filtres.

Une première approche consiste à construire un groupe de filtres qui encapsulent les différentes préoccupations qui doivent être ajoutées aux objets [BA 99]. Pour ajouter du comportement à un objet, il suffit donc de modifier ses filtres d'entrée ou de sortie. Comme les filtres sont généralement orthogonaux les uns aux autres, un ensemble de préoccupations orthogonales peut être appliqué à un objet en ordonnant correctement les filtres.

Dans le cas où les préoccupations ne sont pas toutes orthogonales, le modèle de composition des filtres atteint les limites de son expressivité. Un nouveau filtre doit être défini ; il doit implémenter la composition de toutes les préoccupations. Comme celles-ci ne sont pas orthogonales les unes par rapport aux autres, cette implémentation entrelace fortement les préoccupations. En d'autres termes, cette approche délocalise le tissage des différentes préoccupations vers un filtre auxiliaire.

De plus, comme chaque classe possède un ensemble de filtres qui lui est propre, les comportements à associer à un ensemble de classes ne peuvent être exprimés de manière modulaire. Pour résoudre ce problème le modèle des filtres de composition a été amélioré pour proposer la *super-imposition* [BA 01] qui permet de définir un ensemble de filtres à appliquer sur un ensemble de classes. Cette propriété permet de résoudre le problème d'entrelacement des préoccupations. Une super-imposition est un ensemble de règles qui permet d'énumérer l'ensemble des classes qui doit se voir *imposer* un ensemble de filtres.

La section suivante décrit différentes implémentations du modèle des filtres de composition.

2.4 LANGAGES POUR LA SEPARATION DES PREOCCUPATIONS

Cette section présente un état de l'art des différents langages qui supportent la séparation des préoccupations dans le contexte des langages à classes et dans un environnement compilé, fortement typé. Cet état de l'art est l'occasion de comparer ces différents langages en fonction de leurs aptitudes à séparer les préoccupations. Cette comparaison permet d'établir l'ensemble des langages qui seront étudiés de manière plus approfondie dans le chapitre 3.

2.4.1 AspectJ

AspectJ [ASP 03] est une extension du langage Java [JAV 04] qui supporte les aspects ; elle est issue du modèle proposé par [KLM 97]. AspectJ ajoute en particulier quelques nouvelles constructions syntaxiques dont l'objectif est à la fois de déclarer les aspects, les points de jointures et de fournir un tisseur pour composer les aspects. Ce tisseur produit (dans la version actuelle) du code source Java qui est ensuite compilé automatiquement par le compilateur fourni avec la machine virtuelle Java. Les aspects sont donc composés par une transformation produisant du code Java[1]. La sémantique du langage d'AspectJ propose de voir les aspects comme des classes, et donc de bénéficier des apports du paradigme objet de Java. Quelques ajouts sont cependant proposés :

– Un contrôle assez précis de l'instanciation des aspects : grâce à des constructions spéciales il est possible de préciser si l'aspect doit être instancié une fois par machine virtuelle ou au contraire, chaque fois qu'un objet sur lequel il agit, est créé. L'instanciation des aspects est réalisée uniquement par le système.

[1] Les aspects sont composables vers le langage Java sans l'étendre, au risque de perdre un peu d'expressivité dans les points de jointures mais, en contrepartie, cela permet aux programmes spécifiés avec AspectJ, d'être compatibles avec toute les machines virtuelles Java.

– Une syntaxe très expressive permet de déclarer les points de jointures. Elle permet en particulier de définir un ensemble de méthodes ou d'accès à des champs, ainsi que leur contexte lors de l'appel ou de l'accès. Les points de jointure sont vus comme des données « membres » ; ils ont en particulier la possibilité d'être abstraits, privés, publics, etc.

– Un aspect peut être « privilégié » : toutes ses méthodes ont alors les mêmes droits d'accès que les méthodes des classes sur lesquelles s'applique l'aspect.

– Un aspect peut « dominer » un autre aspect ceci permet l'exécution de cet aspect avant un autre, c'est la règle de composition explicite, la règle implicite étant l'héritage.

– Une construction syntaxique permet d'introduire du code directement dans les classes comme de nouvelles méthodes ou variables, ou même de modifier l'arbre d'héritage en ajoutant de nouvelles interfaces.

De plus, les aspects peuvent aussi être abstraits pour permettre par exemple, la construction de bibliothèques d'aspects réutilisables. L'utilisateur de ces bibliothèques n'aura qu'à concrétiser par héritage les aspects afin de les adapter aux besoins de l'application.

AspectJ est le premier langage basé sur le modèle de la programmation par aspects, c'est donc très probablement celui qui est le plus mature[1]. Il est donc naturel qu'AspectJ apparaisse dans le comparatif du chapitre 3.

2.4.2 Hyper/J

Développé par IBM [HO 93], le *Watson Subject Compiler* est une extension d'un compilateur C++ supportant la programmation par sujets. Il supporte l'intégration de sujets à partir du code source ou à partir d'un nouveau format de fichier binaire contenant des sujets compilés.

Avec Hyper/J, le *Watson Subject Compiler* a suivi l'évolution de la programmation par sujets vers la programmation par HyperSpace dans le but de supporter la séparation multidimensionnelle des préoccupations.

Hyper/J [OT 00a], [HYP 03] est une extension de Java implémentant le modèle des HyperSpaces. Il utilise différents fichiers contenant des HyperSlices et des HyperModules pour produire l'intégration des différents HyperSlices sous la forme de fichiers sources Java. Cette approche permet d'utiliser Hyper/J avec n'importe quelle machine virtuelle Java.

Hyper/J est sûrement le langage le plus abouti pour faire de la programmation par sujets[2]. Il est tout naturellement retenu dans le comparatif proposé dans le Chapitre 3.

2.4.3 Caesar

Caesar [MO 03] est l'implémentation d'un modèle qui a beaucoup évolué comme en témoigne [MO 02], [Ost 02], [HM 01], [Hau 01], [MSL 00] et [LLM 99].

Caesar est une extension de Java supportant la programmation par aspects. Il produit un *byte-code* totalement compatible avec la machine virtuelle de Java. Le modèle sur lequel il s'appuie utilise des « Composants Aspectualisés ». En d'autres termes, cela signifie que ce modèle propose *des interfaces de collaboration encapsulées par la notion d'aspect* : un *aspect* est un ensemble *d'interfaces de collaboration*. Chaque *interface de collaboration* possède des services requis, des redéfinitions d'implémentation et des services locaux.

---

[1] Il est utilisable dans un contexte industriel. Pour plus d'information sur AspectJ se référer à [ASP 03].
[2] Pour plus d'information sur Hyper/J se référer à [OSS 00].

Les interfaces de collaboration, contrairement aux autres langages de programmation par aspects, décomposent l'architecture de l'application suivant plusieurs points de vue : la définition de l'aspect, son implémentation, les connexions entre aspects, et le déploiement des aspects.

Les *aspects* définissent des extensions orthogonales (modélisées par des *interfaces de collaboration*) au reste du système. Chaque *interface de collaboration* peut être considérée comme un Mix-Ins car l'interface a un paramètre générique qui représente la classe à laquelle elle s'applique. Cette notion d'aspect est donc un Mix-In Layer.

L'analogie avec les Mix-Ins prend fin dès que l'on aborde la phase de connexion des aspects avec le reste de l'application. Cette phase est en effet effectuée *en dehors* du code de l'application de manière statique par des connecteurs ; elle permet donc la séparation effective des préoccupations. Ces connecteurs établissent pour chacune des instanciations de l'aspect toutes les liaisons entre les *interfaces de collaboration* et les classes sur lesquelles elles doivent se lier. Ce sont des objets de première classe qui possèdent une relation d'héritage.

La liaison entre les *interfaces de collaboration* et les classes est de type « plusieurs à plusieurs ». Elle est aussi flexible : lors de la définition de la liaison, des adaptations peuvent être effectuées pour mettre correctement en relation les *interfaces de collaboration* et les classes.

Les adaptations possibles concernent essentiellement les services requis des participants. Un service requis d'une *interface de collaboration* est une signature de méthode qu'une liaison peut mettre en relation avec un ensemble de méthodes (qui sont alors exécutées séquentiellement). De plus, la liaison peut aussi permettre de définir des services requis de manière programmative. Il faut alors implémenter une méthode ayant la signature d'un service requis à partir des fonctionnalités offertes par les classes liées.

Les services requis des *interfaces de collaboration* représentent une partie du système des points de jointure du modèle. L'autre partie est localisée dans les connecteurs pour permettre l'exécution de code *avant*, *après* ou *autour* des points de jointure. Notons qu'une décomposition similaire entre point de jointure et méthodes des aspects a aussi été appliquée à AspectJ par [Beu 99].

L'exemple ci dessous est tiré de [MO 03] :

```
1   interface OserverProtocol {
2     interface Subject {
3       provided void addObserver(Observer o);
4       provided void removeObserver(Observer o);
5       provided void changed();
6       excepted String getState();
7     }
8     interface Observer {
9       excepted void notify(Subject s);
10    }
11  }
```

L'aspect `ObserverProtocol` est défini par deux *interfaces de collaboration* `Subject` et `Observer` qui définissent les deux rôles du patron de conception observateur. Ces deux interfaces seront liées à une (des) classe(s) existante(s) au moment de la connexion de l'aspect. Cette connexion mettra en relation des méthodes de la classe avec les méthodes requises (dénotées par le mot clef **`excepted)`**. De plus, elle mettra au service de la classe les méthodes fournies (dénotées par le mot clef **`provided`**).

Les méthodes que les aspects fournissent aux interfaces de collaboration peuvent avoir plusieurs implémentations. Le choix d'une implémentation particulière se fait au moment du déploiement des aspects (c'est-à-dire pendant la dernière phase). Ci-dessous nous proposons une implémentation possible de l'aspect `ObserverProtocol`.

```
12 Class ObserverProtocolImpl implements ObserverProtocol {
13   Class Subject {
14     List observers = new LinkedList();
15     void addObserver(Observer o) { observers.add(o); }
16     void removeObserver(Observer o) { observers.remove(o); }
17     void changed() {
18       Iterator it = observers.iterator();
19       while ( it.hasNext() )
20         ((Observer)iter.next()).notify(this);
21     }
22   }
23 }
```

Pour réutiliser un aspect, il faut définir des connexions entre lui et les classes qui sont concernées. Ci-dessous, on trouve un exemple de connexion de l'aspect `ObserverProtocol`.

```
24 Class ColorObserver binds ObserverProtocol {
25   Class PointSubject binds Subject wraps Point {
26     String getState() {
27       return "Point colored" + wrappe.getColor();
28     }
29   }
30   Class LineSubject binds Subject wraps Line {
31     String getState() {
32       return "Line colored" + wrappe.getColor();
33     }
34   }
35   Class ScreenObserver binds Observer wraps Screen {
36     void notify(Observer s) {
37       wrappe.display("Color changed" + s.getState());
38     }
39   }
40
41   after(Point p) : (call(void p.setColor(Color)))
42     { PointSubject(p).changed(); }
43
44   after(Line l) : (call(void l.setColor(Color)))
45     { LineSubject(l).changed(); }
46 }
```

La connexion d'un aspect permet de fournir ses méthodes requises par lui et de le mettre en relation avec les classes qu'il peut *éventuellement* modifier. C'est le déploiement qui finalement décide des classes à modifier. Cela peut se faire soit statiquement (toutes les instances des classes participantes à la connexion seront alors modifiées par l'aspect), soit dynamiquement, de manière programmative (l'aspect peut ainsi être utilisé pour modifier des instances des classes participant à une connexion).

La connexion `ColorObserver` permet de modifier les classes `Point` et `Line` pour les rendre `Observable` (rôle `Subject`), et la classe `Screen` pour qu'elle se comporte comme un observateur. Une fois les rôles définis, deux points de jointure sont utilisés : un pour `Point` et un autre pour `Line`. Ces points de jointure mettent à jour les observateurs.

Le code ci-dessous présente un exemple de déploiement statique de l'aspect.

```
47  Deploy Class CO extends ObserverProtocol
48                           <ColorObserver,ObserverProtocolImpl> {};
```

Ce déploiement permet de faire la liaison en la connexion `ColorObserver` et l'implémentation `ObserverProtocolImpl` de l'aspect `ObserverProtocol`.

Ainsi, Caesar utilise un modèle de programmation par aspects [KLM 97] (comme AspectJ par exemple). Il possède deux propriétés : l'éclatement total des aspects et le déploiement statique ou dynamique des aspects.

L'éclatement total des aspects consiste à découpler à la fois la définition de l'aspect, son implémentation, ses connexions et son déploiement. Ce découplage présente l'avantage de pouvoir changer, pendant le déploiement, l'implémentation de l'aspect utilisée. Par contre, il diminue sérieusement la lisibilité de ce dernier. En effet, Caesar nécessite que le programmeur regarde au minimum à quatre endroits (plus si l'héritage de connecteurs est utilisé), pour comprendre un aspect. Au contraire, AspectJ ou Hyper/J permettent de fournir toute l'information au même endroit ou de la fractionner selon les besoins de l'utilisateur. Nous pensons que laisser une certaine flexibilité quant à la localisation de la description d'un aspect permet de s'adapter aux règles de programmation mises en oeuvre dans les projets. En supprimant cette possibilité, Caesar va à l'encontre du fonctionnement des projets.

L'implémentation de Caesar n'est pas encore complète, seul le système de type est réalisé. Il semble donc prématuré de le retenir pour le comparatif entre les différentes approches disponibles qui est proposé dans le chapitre 3. Néanmoins, pour satisfaire les lecteurs soucieux d'évaluer plus finement Caesar, nous avons voulu que l'exemple proposé ci-dessus fasse partie des exemples du chapitre 3.

2.4.4 ConcernJ

Le modèle des filtres de composition a été implémenté pour différents langages comme C++ [Gla 95], Java [Wic 99], [Car 01], ou encore Sina [SIN 04].

Parmi ces implémentations, [Gla 95] et [Wic 99] ne sont que des implémentations partielles du modèle des filtres de composition : seul un ensemble restreint de filtres est supporté (*Dispatch* et *Error*), ce qui ne permet pas de bénéficier de tous les avantages du modèle.

De plus, seul ConcernJ [Car 01] supporte la super-imposition[1]. Comme ConcernJ produit du code ComposeJ [Wic 99], il supporte le même ensemble de filtres de composition que ce dernier.

Bien que le modèle des filtres de composition avec la super-imposition apparaît comme prometteur, aucune implémentation actuelle ne met complètement en oeuvre ce modèle. Ainsi, aucune utilisation concrète pour la séparation des préoccupations n'est *a priori* envisageable et donc ce modèle ne sera pas abordé dans le comparatif du chapitre 3.

2.4.5 Java Aspect Component

Java Aspect Component (JAC) [PSD 01], [PSD 01a], et [Paw 02] est un framework pour faire de la programmation par aspects en Java. Par rapport aux langages comme AspectJ, JAC utilise une réification et ne nécessite donc aucune extension du langage lui-même.

Un aspect dans JAC est une instance de la classe `AspectComponent`, il peut, en cours d'exécution d'une application, être déployé ou retiré. Pour cela, JAC utilise Javassist [Chi 00] et

[1] Cette technique permet, de manière modulaire, d'imposer des filtres à des ensembles de classes, plutôt qu'à une classe seulement.

[TSC 01], un protocole à métaobjets pour Java dont l'implémentation repose sur un mécanisme de type *class-loader*.

Trois types d'aspects sont fournis dans ce framework : des décorateurs dynamiques, des objets rôles, et des gestionnaires d'exception. Ils permettent respectivement, de changer le comportement des objets grâce à leur capacité à exécuter du code avant ou après les méthodes décorées, d'ajouter de nouvelles méthodes aux objets[1], et la prise en charge d'exceptions non prévues.

Nous présentons ci-dessous, un exemple issu de la documentation de JAC.

```
1   public class MyWrapper extends Wrapper {
2
3      // a wrapper must call the Wrapper(AspectComponent)
4      // constructor
5      public MyWrapper(AspectComponent ac) {
6          super(ac);
7      }
8
9      // A wrapping method must always have this prototype
10     public Object verboseCall (Interaction interaction) {
11       Object ret = null;
12       System.out.println("<< calling "+interaction.method+
13                          " with "+interaction.args+" >>");
14       ret = proceed(interaction);
15       System.out.println("<< "+interaction.method+
16                          " returned "+ret+" >>");
17       return ret;
18     }
19  }
```

Un décorateur (ou *wrapper*) représente la partie fonctionnelle d'un aspect. Le décorateur `MyWrapper` encapsule une fonctionnalité dédiée aux traces. Il est déployé dynamiquement au sein de l'application grâce au composant d'aspect ci-dessous.

```
20  public class Tracing_1_AC extends AspectComponent {
21      Tracing_1_AC() {
22          pointcut(".*",".*",".*",MyWrapper.class.getName(),
23                   "verboseCall",false,null);
24      }
25  }
```

Le composant d'aspect `Tracing_1_AC` déploie l'aspect `MyWrapper` sur toutes les classes de l'application. Pour plus d'information se reporter à [JAC 03].

JAC est fourni avec un environnement de développement rapide basé à la fois, sur UML et une bibliothèque d'aspects qui couvre divers domaines comme l'authentification, la distribution, les interfaces homme-machine, la persistance, ou la synchronisation.

Néanmoins, JAC ne repose pas sur une extension de langage, il est plus complexe à utiliser qu'AspectJ, et ne fournit aucun moyen supplémentaire par rapport à ce dernier en matière de séparation des préoccupations. C'est pour ces différentes raisons, que malgré ses qualités, JAC ne sera pas inclus dans notre comparatif.

2.4.6 Métaprogrammation logique par aspects

La métaprogrammation logique par aspects (« Aspect-oriented logic Meta Programming ») [VD 98] est d'une part basée sur la représentation des programmes en tant que faits logiques et d'autre part décrit les aspects comme des méta-programmes écrits en langage logique.

[1] Toutefois, ces dernières ne sont pas accessibles directement.

Ce modèle de programmation par aspects diffère du modèle de référence de [KLM 97] par le fait que les points de jointure sont exprimés en langage logique [Gyb 02] et que les prédicats portent sur des métaobjets. Pourtant, nous constatons finalement que les puissances d'expression des deux modèles (ainsi que des diverses implémentations) sont équivalentes. Par ailleurs, nous considérons que d'autres modèles/langages (AspectJ, Hyper/H, etc.) possèdent des systèmes de points de jointure plus simple à appréhender. Comme ces langages seront inclus dans notre comparatif nous n'avons pas jugé utile d'y ajouter cette approche.

2.4.7 Programmation par aspects avec des diagrammes de séquence de messages

La programmation par aspects avec des diagrammes de séquence de messages (*MSC*) [MSC 93] utilise les MSCs comme système de points de jointure [VW 02].

L'avantage de cette approche est d'utiliser le formalisme graphique des MSCs qui permet d'exprimer des points de jointure à l'aide d'un langage visuel. Les MSC permettent dans ce contexte de décrire des séquences d'envois de message (qui sont donc des points de jointure). Quand cette séquence et reproduite par le flot d'exécution du programme l'adaptation associé est réalisée. Le formalisme graphique sous-jacent permet d'appréhender plus facilement le concept de point de jointure. Cette représentation atteint néanmoins une taille limite pour la compréhension quand le point de jointures équivalent fait appel à plusieurs objets en interaction. Elle peut donc nuire à la compréhension. De plus, le langage des MSCs n'est pas assez expressif pour exprimer des adaptations fonctionnelles et de plus, il ne couvre qu'une partie des adaptations comportementales (uniquement l'envoi de message entres des objets parfaitement identifiés). Ceci nous conduit à ne pas inclure cette approche ni ses implémentations dans notre comparatif.

2.4.8 Aspect Moderator Framework.

Aspect Moderator Framework [CSE 01] (AMF) est une bibliothèque pour Java qui permet de réaliser la séparation des préoccupations. Comme *Java Aspect Component* [Paw 02], AMF ne nécessite pas d'extension de langage ; il repose sur un système utilisant le patron de conception du *Proxy* pour adapter le fonctionnement des objets par indirection des messages entrants.

On constate que JAC simplifie l'utilisation des Proxy grâce à une prise en compte par le système de la composition dans son ensemble[1] alors qu'à l'inverse, AMF ne propose qu'un framework partiel pour les Proxy. De plus, JAC propose une bibliothèque d'aspects réutilisables, des outils de conception graphiques et un support pour la distribution. Cette constatation nous amène à ne pas considérer, non plus, AMF dans notre comparatif.

2.4.9 Jiazzi

Jiazzi [MH 03] est une extension de Java pour utiliser du code compilé séparément et le lier en externe. Jiazzi permet grâce à son mécanisme d'ouverture de classe qui fournit un découplage de l'implémentation et des déclarations, de réaliser une séparation des préoccupations. Ce découplage ressemble au découpage qui existe en C ou C++. Avec les fichiers en-tête (suffixe .h) et code (suffixe .c).

Par contre, comme le mécanisme de séparation de Jiazzi permet de séparer uniquement du code qui appartient à une même classe, il n'est pas possible de séparer des préoccupations dont le code est entrelacé avec l'ensemble du système [MH 03].

En d'autres termes, Jiazzi est un langage qui supporte essentiellement la séparation des préoccupations intra classes. Il permet, par exemple, de faire de la programmation par rôles ou par points de vue, mais il ne supporte pas complètement la séparation des préoccupations. A la vue de

[1] On notera cependant que celle-ci peut être surchargée (voir section 2.4.5).

cette limitation nous n'inclurons pas Jiazzi dans notre comparatif détaillé. En effet, un de nos objectifs est de pouvoir séparer les préoccupations qui s'entrelacent avec le reste du système.

2.4.10 JAsCo

JAsCo [SVJ 03] et [JAS 04] est une extension de Java qui supporte une forme de programmation orientée aspects qui est une combinaison entre la puissance d'expression d'AspectJ (voir section 2.4.1) et l'indépendance des aspects de Java Aspect Component (voir section 2.4.5). Cette extension à Java ajoute deux nouvelles entités : les aspects et les connecteurs. Ces nouvelles entités sont décrites de manière programmative.

JAsCo permet de définir des aspects qui peuvent contenir des méthodes et des variables comme les classes, et des *hooks*. Les aspects permettent de réaliser des adaptations de type interception de méthodes, et ces interceptions ont le même potentiel qu'AspectJ. Dans la terminologie du modèle, un *hook* est une sorte de classe interne qui représente une adaptation de type interception. Elle contient un ou plusieurs constructeur qui permettent d'initialiser les cibles de l'adaptation. Le *hook* contient aussi une ou plusieurs *méthodes comportementales* (avant, après, remplacer) qui permettent d'effectuer les adaptations visant à modifier le comportement des cibles.

JAsCo permet aussi d'utiliser des stratégies de combinaison d'aspects pour permettre de résoudre certains problème de composition en ordonnançant l'exécution des aspects en conflits. Elles sont décrite aussi de manière programmative par des classes Java normales qui implémentent juste une interface du modèle.

Enfin, JAsCo permet de définir des *connecteurs*. Un connecteur permet de déployer dynamiquement un ou plusieurs *hooks* dans des contextes spécifiques. Il contient des instanciations des *hooks* qui permettent de fixer les cibles de ces derniers (les constructeurs des hooks prennent toujours en arguments leurs future cibles). Il peut aussi contenir des instanciations de stratégie de combinaison (les constructeurs de ces dernières prennent en argument des *hooks*). Enfin, le connecteur peut appeler un ou plusieurs *méthodes comportementales* des *hooks* qu'il a initialisé. L'ordre d'appel de ces *méthodes comportementales* définit donc l'ordre de composition des aspects.

JAsCo permet donc de définir des aspects indépendamment de leurs futurs contextes de réutilisation. Les connecteurs et les stratégies de combinaison permettent eux de déployer dynamiquement et convenablement les aspects à l'exécution. De plus, JasCo supporte le modèle de composant des *java beans* [JAV 04] en permettant aux *connecteurs* d'instancier des *hooks* sur des évènements propre aux *beans*.

Malgré que JAsCo soit un modèle hybride entre objets et composants, nous n'allons pas le retenir pour la suite car ces capacités d'adaptation sont limitées à l'interception de méthodes. Car nos objectifs nécessitent d'autres types d'adaptation (en particulier des adaptations fonctionnelles) qui sont elles supportées par d'autres approches précédemment citées.

2.5 BILAN.

Cet état de l'art a présenté les différents modèles et langages qui supportent la séparation des préoccupations. Il nous a permis de dégager un certain nombre d'approches dont il semble intéressant de s'inspirer pour appliquer la séparation des préoccupations à l'amélioration de la réutilisation des composants, de leur lisibilité ou de leur capacité à évoluer. Pour chacune des approches retenues, nous proposons dans le chapitre 3 un comparatif plus détaillé sur ces thèmes.

Cette seconde étude a pour vocation de proposer un bilan synthétique qui présente les avantages de la séparation des préoccupations sur des exemples concrets. C'est pour cela, que nous avons choisi de ne pas sélectionner l'ensemble des modèles et langages présentés dans ce chapitre.

Ainsi, seulement quatre modèles (programmation générique, métaprogrammation, programmation par aspects et programmation par sujets) sur les six présentés seront considérés. Deux modèles (programmation par rôles ou point de vue et filtres de composition) ne possèdent aucune implémentation utilisable sur des exemples concrets de séparation des préoccupations.

Le tableau 1 propose à titre de synthèse une classification des langages étudiés en fonction du modèle qu'ils implémentent.

Langage	Modèle associé
AspectJ [ASP 03]	Programmation par aspects [KLM 97]
Hyper/J [OT 00a]	Programmation par sujets [HO 93]
Caesar [MO 03]	Mélange de Composant Aspectualisé [MO 03] et de Programmation par aspects [KLM 97]
ConcernJ [Car 01]	Filtre de composition [ABV 92]
Java Aspect Component [Paw 02]	Programmation par aspects [KLM 97]
métaprogrammation logique par aspects [VD 98]	Programmation par aspects [KLM 97]
programmation par aspects avec des diagrammes de séquence de messages [VW 02].	Programmation par aspects [KLM 97]
Aspect Moderator Framework [CSE 01]	Programmation par aspects [KLM 97]
Jiazzi [MH 03]	Proche du modèle de la programmation orientée rôles ou points de vue
JasCo [JAS 04]	Programmation par aspects [KLM 97]

Tableau 1. *Modèles associés aux différents langages étudiées*

Parmi les quatre modèles sélectionnés, seulement deux (programmation par aspects et programmation par sujets) nécessitent d'étendre un langage pour être pleinement utilisables[1]. Pour ces deux modèles nous avons choisi de prendre leur meilleur représentant, respectivement AspectJ et Hyper/J. Comme l'a montré l'état de l'art ci-dessus, ce sont eux qui ont le meilleur potentiel pour une utilisation réelle ; de plus ils sont des précurseurs pour leurs domaines respectifs.

Pour conclure, le chapitre suivant propose une étude qualitative de la réutilisation de préoccupation pour les quatre modèles sélectionnés et le paradigme de l'objet, en s'appuyant sur des exemples concrets de préoccupations à séparer.

[1] Ceci n'est pas cependant toujours le cas : JAC [Paw 02] et AMF [CSE 01] sont deux contre exemples.

Chapitre 3

Etude qualitative de la réutilisation des préoccupations

Bien que l'approche objet ait introduit l'héritage et le polymorphisme qui sont deux atouts importants pour la réutilisation du code des applications, elle possède dans ce domaine plusieurs limitations [OM 01]. Celles-ci sont exposées dans la suite de ce chapitre.

Dans un premier temps, nous allons étudier la capacité de l'approche objet à intégrer trois types de préoccupations : les préoccupations fonctionnelles et non fonctionnelles et les patrons de conception. Pour chaque type de préoccupation nous montrerons les apports et les limitations des approches par objets. L'état de l'art proposé dans le chapitre 2 a montré qu'il existe des extensions du paradigme de l'objet qui permettent de prendre en compte la séparation des préoccupations mieux que les approches à objets classiques. Nous chercherons donc à évaluer ces approches à savoir, la programmation générique, la métaprogrammation, la programmation par aspects, et la programmation par sujets, en nous appuyant sur les trois types de préoccupations mentionnés ci-dessus.

3.1 LES PREOCCUPATIONS NON FONCTIONNELLES

On désigne sous le terme de *propriétés fonctionnelles* d'une application les services qu'elle fournit (algorithme, logique métier, etc.), qui sont généralement décrits par la spécification d'interfaces. Les *propriétés non fonctionnelles* désignent les caractéristiques de l'application qui sont liées à la façon dont sont implémentées les propriétés fonctionnelles. On peut citer les performances, la fiabilité, la disponibilité, la qualité de service, la distribution, la persistance, le déverminage, etc.

Conceptuellement, les propriétés fonctionnelles et non fonctionnelles concernent des points de vue différents d'une application. Elles sont à la fois, indépendantes de l'application elle-même et indépendantes les unes des autres. On souhaite donc les séparer au niveau de l'implémentation pour pouvoir réutiliser plus facilement chacune de ces préoccupations à savoir, le code fonctionnel et le code non fonctionnel.

La séparation des préoccupations permet de conceptualiser la séparation entre propriétés fonctionnelles et non fonctionnelles sous la forme de préoccupations respectivement fonctionnelles et non fonctionnelles. Nous emploierons donc par la suite la terminologie issue de la séparation des préoccupations pour parler du fonctionnel et du non fonctionnel.

Les préoccupations non fonctionnelles permettent des ajouts de comportements ; ils sont en grande partie indépendants des préoccupations sur lesquelles ils s'appliquent. Ainsi, une préoccupation non fonctionnelle permet d'adapter le comportement d'autres préoccupations, qu'elles soient fonctionnelles ou non fonctionnelles. L'objectif est que celles-ci supportent à leur tour de nouveaux services comme la persistance, la distribution, etc.

Nous étudions dans cette section l'implémentation et la réutilisation d'une préoccupation non fonctionnelle dédiée au traçage de l'exécution d'un programme.

Réaliser des traces est particulièrement utile pour la mise au point de programmes ; néanmoins son utilisation nécessite d'écrire et d'enlever régulièrement les lignes de codes qui s'y rapportent. Même l'utilisation d'une bibliothèque pour faire des traces (par exemple l'API Java *java.util.logging*) alourdit le code source avec des appels aux fonctionnalités de trace. La trace est donc une propriété non fonctionnelle qui peut (doit) être séparée des autres préoccupations.

Plusieurs implémentations de cette préoccupation et leur intégration dans une application sont examinées ; elles s'appuient sur les paradigmes suivants : objet, méta programmation, aspect et sujet. L'étude de ces implémentations va nous permettre de mettre en évidence les apports et les limitations des différents paradigmes. Nous dressons ensuite un bilan sur les propriétés qui sont nécessaires à une « bonne » réutilisation de préoccupations non fonctionnelles.

3.1.1 Préoccupation non fonctionnelle et programmation orientée objet

Pour faciliter la comparaison entre les différents paradigmes, nous avons arbitrairement choisi d'utiliser le langage Java [JAV 04] car AspectJ et Hyper/J que nous allons étudier par la suite sont eux-mêmes des extensions de ce langage.

A titre d'exemple nous proposons d'appliquer la préoccupation non fonctionnelle mentionnée ci-dessus (tracer de l'exécution d'un programme) à une classe représentant une image (elle est nommée `Image`). Comme cette classe a vocation à être utilisée dans différents endroits de l'application et que nous souhaitons tracer l'utilisation de cette classe, quelle que soit la localisation, nous ne pouvons implémenter la trace qu'à l'intérieur de la classe `Image` elle-même.

En effet, indépendamment du langage choisi, le paradigme de l'objet nous offre trois manières d'implémenter la trace :

1. Par sous classage, à travers une classe `ImageAvecTrace` en modifiant les points d'instanciation des images pour instancier des `ImageAvecTrace` (au lieu d'instancier des `Image`).
2. Par délégation en remplaçant les références explicites à `Image` par des références à une classe `proxy` [BFJ 98] et [GHJ 99] qui effectue la trace et délègue ensuite le reste du traitement à la classe `Image`.
3. Par modification directe des méthodes de la classe `Image`.

Notons, que la programmation générique ne permet pas non plus d'envisager d'autre solution que celles mentionnées ci-dessus.

Les solutions (1) et (2) ne sont pas acceptables, car même si elles ont l'avantage de ne pas modifier la classe `Image`, elles obligent en contrepartie à modifier tous les endroits d'utilisation ou d'instanciation de la classe `Image`. Ce problème est connu sous le nom de *problème de la migration des clients*[1]. Ceci n'est donc acceptable que pour des applications de petite taille où le nombre de clients à modifier est restreint.

La solution (2) peut cependant être implémentée par des langages à objets qui supportent la redéfinition des opérateurs et plus particulièrement la redéfinition de l'opérateur de construction des objets, comme par exemple le langage C++. Dans ce contexte, l'opérateur peut être redéfini pour renvoyer une instance du *proxy* (ici la classe `ImageAvecTrace`) au lieu d'une instance de la classe `Image`. Néanmoins, cette solution possède d'autres inconvénients qui sont aussi communs à la solution (3) qui est présentée par la suite.

[1] Le terme anglais est « client migration problem ».

Pour rester indépendant du langage choisi, on ne peut qu'envisager la troisième solution. Nous présentons ci-dessous une implantation en Java qui nous permettra ensuite d'évoquer la réutilisation de la préoccupation.

```
1   public class Image {
2     //...
3     public Image rotation (int angle){
4       //Début de la trace
5       System.out.println("Image.Rotation("+angle+")");
6       //Fin de la trace
7
8       int TailleMax = Math.max(this.largeur,this.hauteur) ;
9       Image tmp = new Image(TailleMax,TailleMax);
10      for(int x=0 ; x<= this.largeur ; x++ )
11        for(int y=0 ; y<= this.largeur ; y++ )
12          tmp.Pixels(Point.Rotation(x,y,angle)) = this.Pixels(x,y);
13
14      //Début de la trace
15      System.out.println("Image.Rotation("+angle+") return : "+tmp);
16      //Fin de la trace
17
18      return tmp;
19    }
20    //...
21  }
```

Cet exemple exhibe la composition (le couplage) entre l'algorithme qui correspond à la logique métier de la méthode et la trace qui est une préoccupation non fonctionnelle possible. Nous analysons à travers cet exemple l'impact de ce couplage suivant deux points de vue : la classe et la préoccupation non fonctionnelle.

La classe, à cause du principe d'encapsulation, incorpore toutes les préoccupations non fonctionnelles associées. La composition des différentes préoccupations est donc faite à la main, le plus souvent avec une granularité correspondant à une méthode. Ainsi, l'implémentation d'une méthode prend en compte les différentes préoccupations pour les composer tandis que l'ajout *a posteriori* de préoccupations nécessite de refaire toute l'étape de composition. L'implantation qui en résulte est difficile à faire évoluer ou même à comprendre à cause du fort couplage qui existe entre les préoccupations. La principale conséquence est de rendre la redéfinition d'une méthode dans une classe héritée extrêmement complexe ; la méthode à redéfinir [MS 98] contient en effet l'ensemble des préoccupations.

On peut dire en d'autres termes que la préoccupation non fonctionnelle disparaît à l'intérieur de la classe sur laquelle elle s'applique. La classe et la préoccupation malgré leur indépendance *à priori*, ne sont plus en fait réutilisables séparément.

Le paradigme objet ne permet pas de séparer les préoccupations non fonctionnelles car elles ne représentent que des extensions comportementales qui sont transversales à la hiérarchie de classes. En effet, même si les différentes préoccupations sont séparées conceptuellement, leur implémentation dans le paradigme objet impose qu'elles soient disséminées dans la hiérarchie des classes.

De plus, la programmation orientée objet ne permet pas d'implémenter des préoccupations non fonctionnelles sans modifier les classes sur lesquelles la préoccupation non fonctionnelle doit s'appliquer. Ceci entraîne un fort couplage entre les préoccupations non fonctionnelles et leurs « clients ».

3.1.2 Préoccupation non fonctionnelle et métaprogrammation

Les métaclasses sont responsables de la sémantique de l'envoi et de la réception des messages. Elles interceptent donc tous les envois de message vers un objet et se chargent de leur exécution. Elles peuvent ainsi modifier le comportement des classes auxquels elles sont associés, tout comme le montre par exemple [Ber 98].

Comme dans la partie précédente, nous allons voir comment réutiliser la même préoccupation non fonctionnelle (tracer l'exécution d'un programme) dans une même classe `Image`. Le protocole à métaobjets permet d'envisager une nouvelle variante par rapport aux trois solutions proposées par le paradigme objet. Elle consiste à délocaliser la composition de la trace et de l'application au niveau méta grâce à la définition d'une métaclasse.

Pour mettre en œuvre cette approche il faut modifier le langage en intégrant une métaclasse qui redéfinit l'appel de méthode pour toutes ses instances (qui sont des classes). Cette nouvelle sémantique de l'appel de méthode doit, en plus de la sémantique normale[1], afficher l'historique (par exemple dans un fichier ou une fenêtre), de tous les appels de méthode qui comprend l'expéditeur, le destinataire et les arguments. Le code source ci-dessous décrit une implémentation possible pour un langage à métaclasses dérivé de Java :

```
1   class MetaClasseTrace extends MetaClass[2] {
2     public Object Invoke (Object from, Object to, Message message) {
3        //Trace
4       System.out.println("Appel de la méthode " +
5          message.MethodName() + " depuis " + from + " vers " + to +
6          "\n avec pour parametres : " + message. Paramètres() + "\n");
7
8       Object resultat = MetaClass.Invoke(from,to,message);
9
10       //Trace
11      System.out.println("Appel de la méthode " +
12         message.MethodName() + " depuis " + from + " vers " + to +
13         "\n avec pour retour : " + resultat + "\n");
14
15       return resultat;
16    }
17  }
```

Dans cet exemple, le MOP est supposé avoir un point d'entrée nommé `Invoke` pour gérer les envois de messages. Cette méthode doit prendre trois paramètres : l'expéditeur, le destinataire et une réification représentant le message à transmettre.

Pour tracer les appels de méthode d'une classe et de ses instances, il faut que cette classe soit une instance de notre métaclasse. Une syntaxe possible pour le langage Java est la suivante :

```
18  public MetaClasseTrace Image{
19     \\...
20  }
```

Notons que dans un langage à métaclasses dynamiques comme CLOS [Kee 89] ou SmallTalk [GR 83], les métaclasses sont présentes pendant l'exécution. Ainsi la préoccupation « trace » pourrait même être ajoutée ou retirée pendant l'exécution.

Cet exemple montre la séparation entre la classe `Image` (la logique métier) et la préoccupation non fonctionnelle (la trace). Comme dans la section précédente, nous revenons sur les points de vue de la classe et de la préoccupation non fonctionnelle.

La classe est quasiment inchangée, seule la métaclasse qui lui correspond a été modifiée pour pouvoir intégrer la préoccupation non fonctionnelle. Nous avons donc utilisé la capacité des métaclasses à s'intercaler entre la réception du message par l'objet destinataire et l'exécution de la méthode comme première approche de séparation des préoccupations [Bou 00]. Ansi la préoccupation a été délocalisée dans les métaclasses.

---

[1] Celle-ci est obtenue par appel de la méthode héritée de la méta classe racine du système.

[2] `MetaClass` est la métaclasse racine de notre système, elle représente la sémantique originale du langage.

Comme la préoccupation non fonctionnelle est réifiée par une métaclasse elle peut être réutilisée pour tracer n'importe quelle classe. Il reste cependant un problème non résolu au niveau des métaclasses. En effet, celles-ci deviennent responsables de l'ajout des nouvelles préoccupations. Or, peu de langages à métaobjets peuvent avoir plus d'une métaclasse par classe [Gra 89].

Cette remarque suggère que la technique utilisée pour appliquer plusieurs propriétés non fonctionnelles à une même classe est dépendante des fonctionnalités du protocole à métaobjets :

– soit le langage accepte plusieurs métaclasses par classe et ainsi il est possible d'encapsuler chaque préoccupation non fonctionnelle dans une métaclasse,

– soit le langage n'offre pas cette possibilité et une seule métaclasse doit donc gérer l'ensemble des préoccupations non fonctionnelles de sa classe associée. Les préoccupations seront alors solidement entrelacées dans la métaclasse.

La première solution permet de réutiliser les préoccupations car celles-ci peuvent être partagées entre plusieurs classes. Nous verrons plus loin dans ce document que ce problème est ouvert car seuls quelques types de composition sont facilement réalisables [MMC 95], [BRL 98], et [BLR 98]. Par contre, la deuxième solution soulève un autre problème : l'utilisateur doit composer à la main la métaclasse contenant toutes les préoccupations qu'il veut appliquer à un même ensemble de classes.

Après l'examen de cet exemple, trois constatations peuvent être faites. D'abord, l'approche par métaprogrammation est plus difficile à maîtriser que d'autres approches (que nous présentons plus loin). Ainsi, celui qui met en oeuvre les préoccupations doit, soit connaître le protocole à métaobjets, soit faire appel à un métaprogrammeur.

Ensuite, l'approche par métaobjets ne permet pas d'exprimer clairement le protocole de composition de la préoccupation. Celui-ci est en effet directement implémenté dans les métaclasses de manière fonctionnelle. Cette propriété réduit fortement la lisibilité du protocole de composition ; cela est d'autant plus vrai que les protocoles à métaobjets sont mal connus.

Enfin, cette approche rend l'aide que les environnements de programmation peuvent apporter à la composition des préoccupations plus difficile à mettre en œuvre[1]. En effet, la présentation à l'utilisateur du code mis à plat, dans lequel toutes les préoccupations ont été composées, devient très complexe à réaliser. Cette vue de l'application après l'étape de la composition est pourtant essentielle car elle permet à l'utilisateur de vérifier plus finement si la sémantique est correcte.

3.1.3 Préoccupation non fonctionnelle et programmation par aspects

Comme dans la section précédente, nous étudions la réutilisation de la même préoccupation non fonctionnelle et de la même classe `Image`. L'état de l'art proposé au chapitre 2 nous conduit à choisir AspectJ [ASP 03] comme langage de référence pour examiner les apports de la programmation par aspects. Pour plus d'information sur AspectJ, se référer aux sections 2.3.4 ou 2.4.1 et aux ouvrages [KLM 97] et [ASP 03].

AspectJ permet d'envisager une nouvelle solution en plus des trois qui sont proposées par le paradigme objet. Elle consiste à encapsuler la préoccupation non fonctionnelle par un aspect, comme le montre [RC 03] ou [Bus 00].

Ainsi, l'implémentation de la préoccupation non fonctionnelle est encapsulée par un aspect abstrait ce qui facilite la réutilisation de la préoccupation comme nous allons le voir par la suite. Cet aspect possède deux méthodes (en fait des *advices*, lignes *6* à *10* et lignes *13* à *18*) qui affichent à l'écran les informations contextuelles du point de jointure. Ce dernier est un élément de la sémantique du langage qui permet à la fois de définir les endroits où déclencher l'exécution des advices, et de transmettre le contexte (voir ligne 3) auquel ils sont attachés.

[1] Par rapport à des approches où la composition est déclarative.

```
1   abstract aspect Trace {
2     // Point de jointure abstrait
3     abstract pointcut classes;
4
5     // Méthode exécutée avant le point de jointure
6     before() : classes() {
7       // demande au point de jointure d'afficher toutes ces
8       // informations contextuelles (message + objet récepteur)
9       System.out.println(thisJoinPoint.toLongString() + "\n");
10    }
11
12    // Méthode exécutée après le point de jointure
13    after() returning o : classes() {
14      // demande au point de jointure d'afficher toutes ces
15      // informations contextuelles (message + objet récepteur)
16      System.out.println(thisJoinPoint.toLongString() +
17        "valeur de retour" + o + "\n");
18    }
19  }
```

L'aspect `Trace` encapsule à la fois le protocole de composition de la trace et son comportement. En effet, nous avons choisi de ne pas délocaliser le code associée à la trace dans une classe normale pour simplifier l'exemple (voir lignes 9, 16, et 17). La préoccupation non fonctionnelle est pleinement réutilisable car l'aspect permet de la séparer de son point d'application et la manière de la composer est déclarée dans l'aspect.

Pour réutiliser la préoccupation dans la classe `Image` il faut spécialiser par héritage l'aspect `Trace` pour concrétiser le point de jointure `classes`. Cette concrétisation permet de compléter la définition du point de jointure en spécifiant ce qui doit être tracé pour l'application (ici, tracer les appels de toutes les méthodes de la classe `Image`).

```
20  aspect TraceMonImage extends Trace {
21    // Spécialisation du point de jointure hérité
22    // pour se fixer à l'exécution des méthodes de la classe Image
23    // avec des arguments et un type de retour quelconques
24    pointcut classes : execution ( * Image.* (..));
25  }
```

Une autre spécialisation possible est décrite ci-après ; elle propose de ne tracer que les appels aux méthodes de la classe `Image` dont le nom commence par `Rotation`.

```
26  aspect TraceMonImage extends Trace {
27    // Spécialisation du point de jointures héritée
28    // pour se fixer aux méthodes dont le nom commence par Rotation
29    // avec des arguments et un type de retour quelconques
30    pointcut classes : execution ( * Image.Rotation* (..));
31  }
```

On constate donc qu'AspectJ a permis de séparer la classe `Image` qui représente la logique métier de la préoccupation non fonctionnelle qui implémente une fonction de « trace ».

Grâce au modèle de composition d'AspectJ le code source des classes n'est pas modifié directement par le programmeur. Le fait que cette composition soit *in-situ* sous-entend que les classes associées sont modifiées de manière transparente, c'est-à-dire, sans créer de nouvelles versions de ces classes.

La préoccupation non fonctionnelle est réifiée par un aspect et ne décrit pas explicitement les entités de l'application à laquelle elle est rattachée ; elle devient donc aisément réutilisable pour

tracer n'importe quelle classe (un simple sous-classage suffit à décrire l'adaptation souhaitée) et son évolution est indépendante du reste de l'application.

Notons, que l'approche par aspects est plus simple que l'approche par métaobjets car celui qui programme les préoccupations doit maîtriser uniquement les opérateurs de composition fournis par le système de points de jointure. Les points de jointure ne représentent que des entités bien ciblées du langage (classes, méthodes et variables d'instances) alors que les métaclasses encapsulent toute la sémantique du langage.

L'exemple ci-dessus peut être généralisé et toute autre préoccupation non fonctionnelle peut être exprimée par un aspect. Cependant il faut alors pouvoir réaliser la composition de cet ensemble de préoccupations sur un même ensemble de classes.

La difficulté de composition apparaît lorsque des aspects différents possède des points de jointures dont l'intersection n'est pas nulle. Dans de nombreux cas, les adaptations correspondant aux aspects « en conflit » conduisent à des résultats différents suivant l'ordre d'exécution des aspects.

Pour illustrer ce problème, considérons une application qui réalise des calculs coûteux dont les résultats sont confidentiels. Cette application dispose de deux aspects non fonctionnels qui sont l'optimisation et la sécurité. L'optimisation consiste à utiliser une antémémoire (*cache*) où sont stockés les résultats de calculs. La sécurité consiste à authentifier les demandeurs de calculs, avant de réaliser les traitements. La composition de ces deux aspects, optimisation et sécurité, est critique. En effet, il est impératif que l'authentification du demandeur d'un calcul ait lieu avant d'extraire le résultat du calcul de l'antémémoire. Autrement dit, la composition des deux aspects sécurité et optimisation n'est pas commutative, puisqu'il est impératif d'ordonner les aspects de manière à *privilégier* l'aspect sécurité. Cependant, les aspects étant définis indépendamment les uns des autres, aucune information sur leur composition n'est disponible. Dès lors, l'outil de composition ne dispose pas d'assez d'informations pour déterminer le résultat attendu par les développeurs.

Pour tenter de résoudre ce problème, quelques mises en oeuvre de la programmation par aspects permettent aux développeurs d'étendre les définitions des aspects avec une information qui permet à l'outil de composition de décider de l'aspect à *privilégier* (c'est le cas d'AspectJ). Cependant, cette solution ne permet que de restreindre le champ d'apparition du problème de composition des aspects, sans l'éliminer complètement.

En effet, la définition d'une relation d'ordre partiel entre les aspects résout le problème de composition uniquement si chaque aspect de l'ensemble des aspects à composer sont *orthogonaux*[1] [Gra 89] (ou encore *faiblement non-orthogonaux*[2]) deux à deux. Ce problème date de bien avant la programmation par aspects, il est apparu dans les système à métaclasse qui permettent l'utilisation de plusieurs métaclasses pour une même classe [Gra 89]. Des solutions permettent d'ordonnancer automatiquement les métaclasses en conflit existent [BLR 98] [BRL 98] mais elles réduisent fortement les capacités d'adaptation des meta-classes.

Par contre, la composition de préoccupations *non-orthogonales* est impossible car leurs interactions deviennent trop complexes pour être traitées par des opérateurs d'adaptations (quelle que soit l'approche choisie : aspects, sujets, métaprogrammation, etc.). Une préoccupation unique doit représenter cet ensemble de préoccupations. Les préoccupations seront composées à la main dans une préoccupation, pour ensuite être composé de manière conventionnelle avec le reste du programme. Avec pour résultat de *délocaliser* la composition d'un ensemble de préoccupations *non-orthogonales* à l'intérieur d'une seule préoccupations a pour résultat de rendre cet ensemble de

[1] Deux aspects (ou préoccupations) sont dits orthogonaux s'il n'existe aucune interaction entre eux.
[2] Deux aspects (ou préoccupations) sont dits faiblement orthogonaux s'il leurs interactions nécessitent juste un ordonnancement pour permettre leur composition.

préoccupations non évolutif (car les différents comportements qui le composent sont tissés à la main).

3.1.4 Préoccupation non fonctionnelle et programmation par sujets

Dans cette partie nous reprenons le même exemple mais avec la programmation par sujets. L'état de l'art du Chapitre 2 a guidé notre choix vers Hyper/J [OT 00a] qui sera le langage de programmation par sujets que nous utilisons ci-après. Pour plus d'information sur Hyper/J voir les sections 2.3.5 ou 2.4.2 et à [HO 93] ou [OT 00a].

Nous rappelons ici qu'un sujet est une unité de regroupement de classes au même titre que la notion de package dans Java. La préoccupation « trace » est encapsulée dans un sujet appelé `NonFonctionnel.Trace` tandis que la classe `Image` est encapsulée dans un second sujet appelé `CoucheMetier.Image`. Voici la définition du premier sujet :

```
1   package utils.debug;
2   public class Trace
3   {
4      public static void beforeInvoke ( String classe,
5                                        String methode )
6       {
7         System.out.println(classe + "." + methode);
8       }
9   }
```

La classe `Trace` encapsule une version minimaliste de la préoccupation dédiée aux traces de programme que nous souhaitons réutiliser dans notre exemple.

La fonctionnalité de trace doit aussi être composée par l'intermédiaire d'un module de composition avec le reste de l'application. L'HyperModule (voir 2.3.5 et 2.4.2) suivant permet d'intégrer la trace avec la classe `Image` afin de tracer tous les appels de méthodes de la classe `Image` (lignes 20 à 21).

Il faut maintenant mentionner que la préoccupation non fonctionnelle fait partie du sujet `NonFonctionnel.Trace` (ligne 10). Il en va de même pour la classe `Image` (ligne 11). C'est l'étape d'identification des sujets, qui permet de restructurer éventuellement l'application (voir [OHBS 94]).

Le module de composition décrit ci-dessous utilise un opérateur de composition : **`Bracket`**. Sa syntaxe est la suivante : **Bracket** *méthode1* **with before** *méthode2*. Cet opérateur de composition comportemental provoque l'exécution de la *méthode2* avant un appel à la *méthode1* sans transporter de contexte (paramètres, valeurs de retour, etc.) de la *méthode1* vers la *méthode2*. Le seul contexte fourni par Hyper/J correspond à deux pseudos-variables : `$ClassName`, `$Operation-Name`, elles représentent respectivement le nom de la classe et le nom de la méthode.

Tous les opérateurs de composition d'Hyper/J sont *ex-situ* [OK 00], c'est-à-dire que la composition à laquelle ils participent est produite dans un nouveau sujet. Ce dernier possède le même nom que le module de composition qui l'a produit.

```
10  NonFonctionnel.Trace : package utils.debug
11  CoucheMetier.Image: package couchemetier.image
12
13  hypermodule TracePourImage
14     hyperslices:
15        NonFonctionnel.Trace,
16        CoucheMetier.Image;
17     relationships:
18        mergeByName;
19
20        bracket "Image"."*" with
21         before Trace.beforeInvoke( $ClassName, $OperationName );
22  end hypermodule;
```

Le module de composition permet d'appliquer la préoccupation non fonctionnelle à l'application. La granularité est variable, elle peut aller de plusieurs classes à une méthode d'une classe ou encore à une instance de classe, ce qui représente la granularité minimale. L'information contextuelle transportée par les jointures est très pauvre[1] (nom de la classe, nom de la méthode) ; ce manque d'information contextuelle est dû au fait que dans le paradigme des sujets, la méthode de composition s'appuie sur la fusion de classes. Hyper/J a donc naturellement une faible expressivité en ce qui concerne les opérateurs d'adaptation comportementaux.

Le module de composition a donc permis de séparer la préoccupation non fonctionnelle de la classe `Image` (logique métier). Ceci est possible pour deux raisons : *i)* la préoccupation non fonctionnelle a pu être exprimée par une classe découplée du reste de l'application et *ii)* un module de composition a permis de la composer avec ses classes cibles. La préoccupation non fonctionnelle peut évoluer plus facilement car elle est découplée du reste de l'application. Il faut cependant dans Hyper/J souligner l'absence de relation d'héritage ou d'abstraction pour les modules de composition. Cela rend impossible l'encapsulation d'un protocole de composition dans un module abstrait que l'utilisateur n'aurait qu'à concrétiser ; la composition est donc rendue plus difficile car l'utilisateur n'est pas guidé par un protocole lors de la composition.

Une autre difficulté pour le programmeur est que même si la classe `Image` reste inchangée, il subsiste le problème de la migration des clients. En effet, les clients de la classe `Image` doivent maintenant utiliser la classe produite par le module de composition. L'utilisation d'Hyper/J est donc à ce titre plus difficile que les modèles cités précédemment.

En effet, le paradigme des sujets nécessite que toutes les autres classes du programme soit intégrées dans un ou plusieurs sujets. Ces derniers doivent ensuite être composés avec le sujet produit par le module de composition `TracePourImage` afin d'utiliser la classe `Image` avec la trace. Cet inconvénient provient du fait qu'Hyper/J possède uniquement un modèle de composition *ex-situ* où toute composition produit de nouveaux sujets[2]. Ceci est a l'opposé de la programmation par aspects dont le modèle de composition est *in-situ* (toute composition modifie directement ses classes clientes).

L'utilisation d'Hyper/J introduit une phase préalable au travail d'implémentation et de réutilisation de la préoccupation ; elle consiste à réaliser une succession de mises à jour sur l'application. En effet, les parties de l'application qui doivent utiliser des sujets doivent à leur tour être transformées en sujet afin de pouvoir être elles-mêmes composées avec les sujets qu'elles utilisent (ceci est bien évidement récursif). En d'autres termes, cela signifie que l'utilisation de la programmation par sujets doit être étendue à l'ensemble de l'application et non pas seulement aux préoccupations non fonctionnelles.

[1] Ce manque de contexte des points de jointure est dû uniquement au compilateur de sujets Hyper/J.
[2] Hyper/J ne peut pas, par exemple, modifier les sujets qui interviennent dans la composition.

On constate enfin, dans le code présenté ci-dessus, que la préoccupation « trace » ne possède pas toutes les fonctionnalités attendues. En effet, les valeurs de retour des fonctions tracées sont inaccessibles à cause du peu d'information contextuelle associée aux points de jointures.

Malgré les nombreux inconvénients mentionnés ci-dessus, l'approche par sujets reste intéressante et plus abordable que l'approche par métaobjets pour les mêmes raisons que celles évoquées dans la section 3.1.3 (programmation par aspects).

Comme dans le cas de la programmation par aspects, l'ensemble des préoccupations non fonctionnelle ne peut être exprimé par un seul sujet, d'où la nécessité de les composer.

Les remarques présentes dans la section 3.1.3 sur les problèmes de composition des préoccupations s'appliquent aussi à la programmation par sujets. La seule différence avec la programmation par aspects et que la possibilité d'ordonnancer les sujets et une propriété implicite des modules de composition car chaque module de composition produit un nouveau sujet. Pour ordonnancer un module de composition après un autre, il suffit que le deuxième importe le sujet produit par le premier.

3.1.5 Bilan sur les préoccupations non fonctionnelles

Le non fonctionnel est l'ensemble des préoccupations qui fournissent des services au reste de l'application. Les aspects non fonctionnels peuvent concerner l'ensemble ou seulement une partie de l'application ; ils représentent des extensions comportementales qui sont transversales à l'application. Nous avons choisi de nommer « préoccupation non fonctionnelle » tout élément non fonctionnel qui se retrouve dans cette définition.

Les préoccupations non fonctionnelles doivent être séparées du reste de l'application, c'est-à-dire des éléments auxquels elles s'appliquent. L'étude menée dans cette section vise à répondre à trois questions : *i)* ces préoccupations sont-elles séparables ? ii) si c'est le cas, comment les séparer ? et iii) est-il facile de réutiliser les préoccupations non fonctionnelles ainsi séparées ? Cette étude se voulant pragmatique, elle s'appuie sur un exemple concret : la préoccupation « trace ». Aussi trivial soit-il, cet exemple met en évidence un ensemble de problèmes que nous récapitulons ci-après.

La section 3.1.1 a montré que l'objet (avec ou sans support de la généricité) ne permet pas de séparer les préoccupations non fonctionnelles des classes concernées. La principale cause est le principe d'encapsulation qui demande que les classes concernées par les préoccupations non fonctionnelles les incorporent toutes ; les préoccupations non fonctionnelles sont donc fortement couplées aux classes [MS 98].

La section 3.1.2 a montré que la métaprogrammation [KRB 91] permet de séparer les préoccupations non fonctionnelles de leur composition avec le reste de l'application qui est délocalisée au niveau méta. L'étude a cependant permis de constater que cette solution a des limitations par rapport à la réutilisation des préoccupations. Ainsi la métaprogrammation introduit une plus grande complexité et par voie de conséquence rend la réutilisation des préoccupations non fonctionnelle elle aussi plus délicate. La principale cause de cette situation est l'implémentation manuelle (par le programmeur) du protocole de composition des préoccupations non fonctionnelles dans les métaclasses.

La section 3.2.3 a montré que la programmation par aspects [KLM 97] et plus particulièrement AspectJ [ASP 03] permet de séparer les préoccupations non fonctionnelles. La réutilisation de la partie non fonctionnelle est facilitée pour plusieurs raisons : *i)* il est possible d'utiliser un aspect abstrait pour encapsuler le protocole de composition d'une préoccupation non fonctionnelle, ii) réutiliser la préoccupation revient à spécialiser par héritage l'aspect abstrait pour le concrétiser ; cette concrétisation utilise une approche par points de jointure qui est facilement compréhensible.

La section 3.1.4 a montré que la programmation par sujets [HO 93] et plus précisément Hyper/J [OT 00a] permet de séparer les préoccupations non fonctionnelles. Cependant, la réutilisation de la partie non fonctionnelle n'est pas aisée pour plusieurs raisons. Premièrement il n'est pas possible d'encapsuler des protocoles de composition et l'utilisateur doit donc récrire sans aide toutes les opérations de composition qui sont nécessaires à chaque préoccupation. Deuxièmement, il n'est pas possible d'utiliser la programmation par sujets seulement pour une partie d'une application, l'utilisateur doit en effet l'utiliser pour développer l'ensemble de son application.

En conclusion, on dira que seule la programmation par aspects permet de séparer et de réutiliser facilement des préoccupations non fonctionnelles.

3.2 PREOCCUPATIONS FONCTIONNELLES

Les préoccupations fonctionnelles correspondent aux fonctionnalités d'une application. En d'autres termes il s'agit des éléments de l'application qui la décrivent et en particulier sa logique métier, c'est-à-dire la représentation et le traitement de l'information directement en relation avec l'objectif principal de l'application. Elles s'opposent aux préoccupations non fonctionnelles qui viennent se rajouter *a priori* ou *a posteriori*, à la description des préoccupations fonctionnelles pour lesquelles elles représentent des services additionnels.

Bien que les préoccupations fonctionnelles semblent plus dépendantes de l'application, elles peuvent être partagées par un ensemble d'applications. La logique métier des applications est un exemple de préoccupations fonctionnelles ; on regroupe souvent sous ce terme l'ensemble des préoccupations fonctionnelles d'une application.

Par exemple, dans le contexte du traitement informatique du « workflow », la gestion de stock est un autre exemple de préoccupation fonctionnelle. C'est une fonction commune à un ensemble d'applications, par exemple : la gestion de commerce, la banque (gestion de portefeuilles boursiers), etc.

Par exemple, dans le contexte de l'approche MDA [MDA 04], les développements dirigés par le domaine et la logique métier de l'application correspondent exactement à ces préoccupations fonctionnelles.

De plus, les préoccupations fonctionnelles sont comparables aux composants [Szy 98] car ils représentent tous deux des techniques d'encapsulation différentes de la logique métier.

Pour mettre en évidence les besoins spécifiques liés aux préoccupations fonctionnelles, nous proposons à titre d'exemple d'en considérer trois qui participent toutes à la description de la logique métier d'un compilateur. Ainsi, tous les compilateurs manipulent un arbre de syntaxe abstraite. Celui-ci représente un ensemble de préoccupations fonctionnelles qui mettent en oeuvre la structure arborescente, la grammaire de la syntaxe abstraite, et un ensemble de calculs.

Si les diverses préoccupations formant un arbre de syntaxe abstraite pouvaient être séparées, chacune gagnerait en réutilisation. C'est pour mesurer cette capacité de réutilisation en fonction de l'approche choisie que nous étudions la mise en œuvre de cet exemple avec les approches sélectionnées dans la section 2.3, à savoir les approches par objets, par métaprogrammation, et orientées aspects, ou sujets.

La structure de l'arbre des expressions arithmétiques est encapsulée par un ensemble de classes nommé `expr.syntaxe` (voir figure 2 ci dessous).

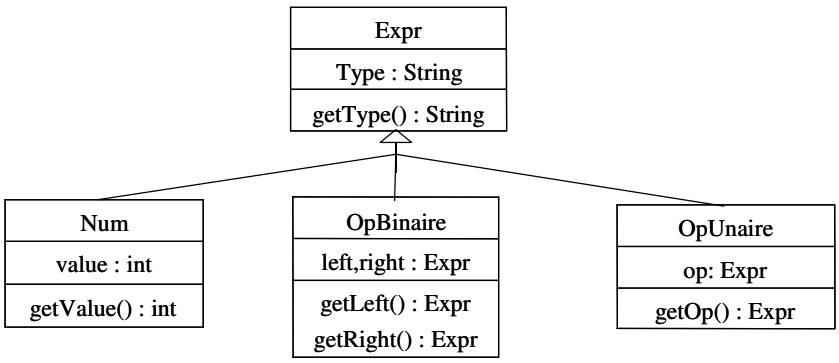


Figure 2. *Le paquetage expr.syntaxe encapsule la structure de l'arbre.*

L'évaluation d'expressions (voir figure 3) arithmétiques est encapsulée par un ensemble de classes nommé `expr.eval`. On remarque qu'il est un sous-ensemble de la structure `expr.syntaxe` car la hiérarchie de classes est la même.

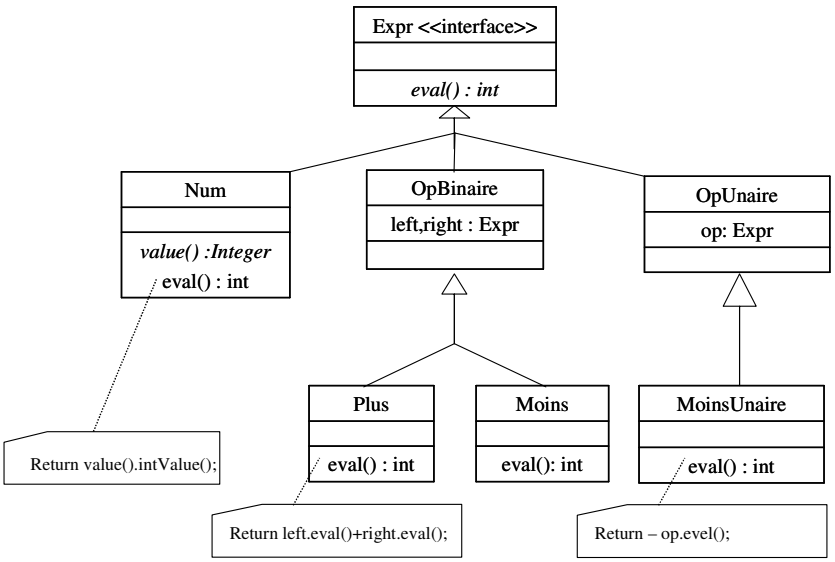


Figure 3. *Le paquetage expr.eval encapsule l'évaluation d'expression arithmétique.*

L'affichage de structures arborescentes (voir figure 4) est fourni par un ensemble de classes nommé `pattern.affichageast`.

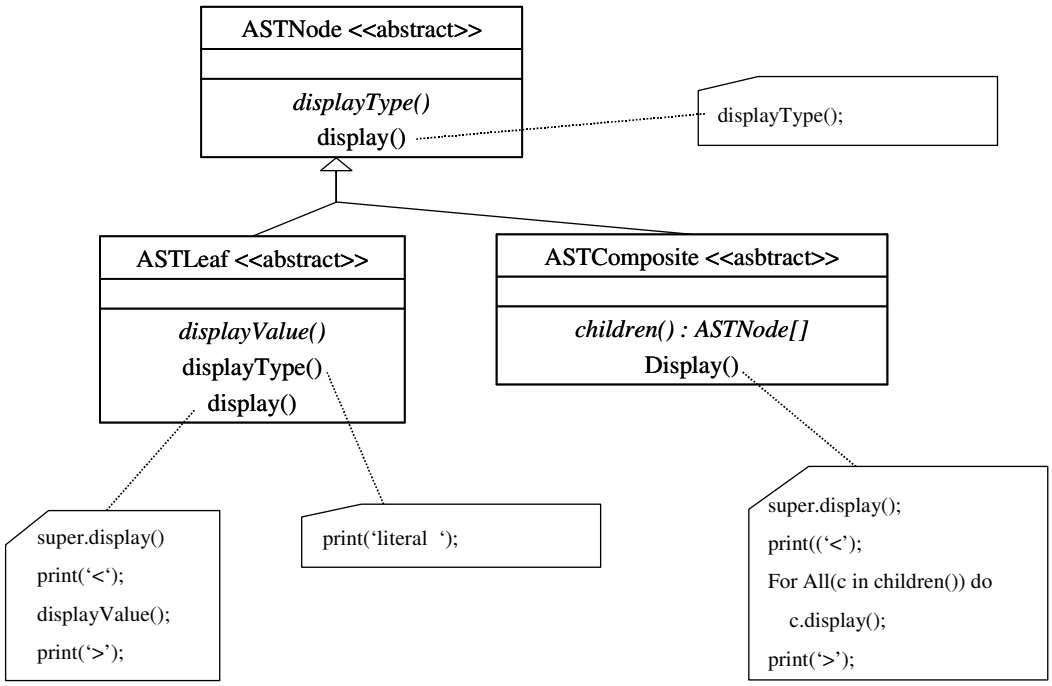


Figure 4. *Le paquetage pattern.affichageast encapsule l'affichage de structures arborescentes.*

Considérer ces trois préoccupations de manière indépendante permet de les concevoir et de les implémenter plus facilement mais aussi de faire en sorte qu'initialement aucun couplage n'existe entre elles. La suite de cette section va permettre d'évaluer comment les différents paradigmes mentionnés ci-dessus parviennent à composer ces trois préoccupations pour aboutir à un arbre de syntaxe d'expressions arithmétiques qui a la capacité d'être évalué et affiché sur un support visuel.

3.2.1 Préoccupation fonctionnelle et programmation orientée objets

Pour réaliser la fonctionnalité décrite ci-dessus, le paradigme de l'objet nous propose quatre solutions :

1. Modifier `expr.syntaxe` pour intégrer l'évaluation et l'affichage.
2. Modifier `expr.affichage` pour intégrer l'affichage et les éléments de `expr.syntaxe` qui sont absents de `expr.affichage`.
3. Utiliser l'héritage multiple pour intégrer dans `expr.affichage` des relations d'héritage vers les classes contenues dans `expr.syntaxe` et dans `pattern.affichageast`.
4. Utiliser la notion d'interface en plus de celle des classes, l'héritage simple de classe et l'héritage d'interfaces pour intégrer dans `expr.syntaxe` les éléments contenus dans `expr.affichage` et dans `pattern.affichageast`.

Les solutions (1) et (2) imposent une duplication massive de code et les trois préoccupations fonctionnelles ne sont plus séparées ! Ainsi les trois préoccupations initialement séparées n'auront servi que de patron préalablement à la réalisation de la fonctionnalité demandée. Toute modifica-

tion d'une des trois préoccupations ne peut être prise en compte de manière automatique dans l'application ainsi constituée.

La solution (3) permet de composer ensemble les trois préoccupations moyennant quelques adaptations rendues obligatoires par des définitions redondantes comme par exemple : les variables d'instances de `OpBinaire` qui sont présentes dans les packages `expr.affichage` et `expr.syntaxe`. Cette solution oblige donc la modification des trois ensembles de classes et donc, une fois qu'elle est réalisée, la séparation entre les trois préoccupations est là aussi perdue. Ainsi ils ne peuvent que difficilement être réutilisés dans d'autres applications. L'héritage multiple [Har 01] n'est pas utilisable dans ce cas.

La solution (4) nécessite de modifier les paquetages `expr.affichage` et `pattern.affichageast` pour à la fois transformer en interfaces les classes abstraites et délocaliser leurs méthodes concrètes dans les classes équivalentes du paquetage `expr.syntaxe`.

Quelle que soit la solution choisie, il est nécessaire de modifier au moins une des trois préoccupations ! Ceci empêche toute réutilisation future des préoccupations modifiées, excepté en dupliquant une des préoccupations mais ceci bloque alors l'évolution de cette dernière. En conclusion, il faut admettre qu'aucune des solutions envisageables avec l'approche par objet, n'est véritablement acceptable.

3.2.2 Préoccupation fonctionnelle et métaprogrammation

La métaprogrammation propose essentiellement d'utiliser la capacité de « génération de code » pour pouvoir produire un nouvel ensemble de classes qui met en œuvre les trois préoccupations mentionnées au début de la section 3.2.

Cette solution nécessite un protocole à métaobjets doté d'une capacité de réflexion qui permette à la fois une introspection dont la granularité doit atteindre les instructions contenues dans les méthodes, et une intercession capable de générer de nouvelles classes à la volée. La mise en œuvre de cette solution est extrêmement difficile car, d'une part, elle demande une maîtrise complète du protocole à métaobjets et, d'autre part, l'implémentation dépend complètement des préoccupations fonctionnelles qui doivent être composées. Le travail fourni pour mettre en œuvre cette solution est dépendant de l'ensemble des préoccupations à composer, il n'est donc pas réutilisable. La métaprogrammation n'est pas une approche qui permet sérieusement d'envisager la réutilisation de préoccupations fonctionnelles.

3.2.3 Préoccupation fonctionnelle et programmation par aspects

Comme cela a été évoqué dans la section 2.3.4, le paradigme des aspects propose d'adapter les classes d'un programme à l'aide d'une entité appelée « aspect ». Par ailleurs, les deux sections précédentes ont montré que pour pouvoir envisager la réutilisation des préoccupations fonctionnelles, celles-ci ne devaient pas être modifiées.

La nature *in-situ* (modification directe des classes) de la composition fournie par la programmation par aspects et ses différentes mises en oeuvre limitent sa capacité à composer des préoccupations fonctionnelles. Ainsi, la seule solution envisageable consiste à modifier les classes des paquetages `expr.syntaxe` et `pattern.affichageast` pour les transformer en interfaces (ou en classes abstraites pures selon les capacités du langage). Ces interfaces pourront dans un deuxième temps être concrétisées avec des aspects pour leur redonner l'implémentation qui leur a été retirée par la transformation des classes en interfaces.

Dans un troisième temps, un aspect supplémentaire permettra de modifier les classes du paquetage `expr.eval` afin d'implémenter l'évaluation d'expressions dans les différentes interfaces (rendues concrètes) des paquetages `expr.syntaxe` et `pattern.affichageast`.

L'inconvénient majeur de cette approche apparaît lorsque les mêmes préoccupations fonctionnelles doivent être utilisées à plusieurs endroits d'une même application. En effet, dans ce cas la programmation par aspects ne permet pas de différencier les différentes réutilisations (car le paradigme des aspects utilise un mode de composition uniquement *in-situ*) de ces préoccupations et donc, *a fortiori*, de les adapter différemment.

On constate donc que les classes du paquetage `expr.eval` sont modifiées par la composition et que leur réutilisation dans d'autres contextes (à l'intérieur d'une même application) doit tenir compte des spécificités de cette composition. Symétriquement, si d'autres compositions faisant intervenir d'autres préoccupations fonctionnelles nécessitent la modification des classes de ce même paquetage alors la composition proposée devra aussi être adaptée.

La programmation par aspects, à cause de sa politique de composition uniquement *in-situ*, ne permet pas d'envisager des solutions viables au problème de réutilisation des préoccupations fonctionnelles sauf si elles ne sont utilisées qu'une seule fois par application. Selon nous cette restriction est beaucoup trop limitative. De plus, comme les adaptations fonctionnelles des aspects ne peuvent pas toutes s'abstraire, il est difficile d'encapsuler les protocoles de composition des préoccupations fonctionnelles.

3.2.4 Préoccupation fonctionnelle et programmation par sujets

La programmation par sujets permet d'envisager une solution au problème de la réutilisation des préoccupations fonctionnelles, car une des propriétés de tous les opérateurs d'adaptation qu'elle propose est la composition *ex-situ*. Une composition *ex-situ* d'un ensemble de sujets produit toujours un nouveau sujet. Cette propriété permet au paradigme des sujets d'être un bon support pour de multiples réutilisations des mêmes préoccupations fonctionnelles dans une même application ou dans plusieurs applications.

Pour étudier les apports de la programmation par sujets, nous avons choisi d'utiliser Hyper/J pour produire un nouveau sujet composite à partir des trois préoccupations fonctionnelles proposées dans les sections précédentes (structure, évaluation et affichage d'un arbre). Ce nouveau sujet pourra ensuite être utilisé par l'application.

Pour cela nous devons d'abord définir des alias pour les ensembles de classes qui correspondent à nos trois préoccupations (lignes 1 à 4). Ces alias vont permettre de réaliser l'union de ces ensembles initialement disjoints.

Pour réaliser la composition des trois préoccupations il a été nécessaire d'utiliser un nouvel ensemble de classes `temp.adaptation.expr` qui permet, comme cela va être démontré par la suite, de résoudre plusieurs problèmes de composition qu'il n'est pas possible de traiter avec les opérateurs d'adaptation fournis par Hyper/J.

Le module `ExpressionArithmetique` qui sert de support à la composition utilise l'opérateur **equate class** qui possède la syntaxe suivante : **equate class** *classe1 classe2*. C'est un opérateur d'adaptation fonctionnel qui permet de fusionner les *classe1* et *classe2* à l'intérieur d'une nouvelle classe dont le nom est *classe1* et qui est produite par le module de composition. La fusion de classes est une opération récursive qui s'applique aussi aux méthodes et aux variables. L'utilisation de l'opérateur **mergebyname** (ligne 13) est un raccourci qui assure la fusion de toutes les classes possédant le même nom.

Le code proposé ci-dessous permet de produire le nouveau sujet composite évoqué dans les lignes précédentes.

```
1   Expr.Syntaxe : package expr.syntaxe;
2   Expr.Eval    : package expr.eval;
3   Pattern.affichageast : package pattern.affichageast;
4   Temp.Adaptation  : package temp.adaptation.expr ;
5
6   hypermodule ExpressionArithmetique
7     hyperslices :
8       Expr.Syntaxe,
9       Expr.Eval,
10      Pattern.affichageast,
11      Temp.Adaptation ;
12    relationships :
13      mergebyname;
14
15      equate class Expr.Syntaxe.Expr , ASTNode;
16      equate class Expr.Syntaxe.Num  , ASTLeaf;
17      equate class Expr.Syntaxe.OpBinaire , ASTComposite;
18      equate class Expr.Syntaxe.OpUnaire  , ASTComposite;
19
20      equate class  Expr.Syntaxe.Expr ,
21                    Temp.Adaptation.ExprAffichage ;
22
23      equate class  Expr.Syntaxe.Num ,
24                    Temp.Adaptation.NumAffichage ;
25
26      equate class  Expr.Syntaxe.OpBinaire ,
27                    Temp.Adaptation.OpBinaireAffichage ;
28
29      equate class  Expr.Syntaxe.OpUnaire ,
30                    Temp.Adaptation.OpUnaireAffichage ;
31  end hypermodule;
```

Le module de composition `ExpressionArithmetique` produit un nouveau sujet (qui porte le même nom) qui représente les expressions arithmétiques affichables et évaluables. Ce nouveau sujet est produit à partir des ensembles de classes correspondant aux trois préoccupations qui ont été encapsulés dans des sujets (voir lignes 1 à 3).

La règle de composition utilisée est la fusion par nom identique (voir ligne 13). Elle permet de composer les paquetages `expr.syntaxe` et `expr.eval` sans avoir à utiliser d'autres opérateurs de composition car les classes qu'ils contiennent, et qui correspondent aux même entités, portent le même nom.

Cependant, pour composer le paquetage `pattern.affichageast` nous devons faire les correspondances « à la main » car les classes à fusionner portent des noms différents. Nous avons ainsi utilisé l'opérateur **`equate class`** (lignes 15 à 18), qui permet d'identifier un ensemble de classes à fusionner. La classe qui résulte de la fusion prend le nom de la liste associée à l'opérateur **`equate`**. Par exemple la ligne 16 compose la classe `ASTLeaf` dans la classe `Expr.Syntaxe.Expr`.

A titre de synthèse, on peut dire que la fusion d'une collection ordonnée de classes est donc une opération d'adaptation qui intègre une nouvelle classe à l'intérieur du sujet produit par le module de composition. Cette nouvelle classe contient les éléments des classes ainsi fusionnées et porte le même nom que la première classe de cette collection. Le fait de choisir une politique générale de composition basée sur l'égalité de nom (**`mergebyname`**) implique que lorsque deux éléments (méthodes, variables d'instance ou variables de classe) ont le même nom, ils sont automatiquement fusionnés. Dans le cas où deux éléments qui ne portent pas le même nom doivent quand même être fusionnés, il faut utiliser l'opérateur **`equate`** suivit des éléments à fusionner.

Pour revenir à ce qui a été mentionné plus haut, la composition des trois paquetages a nécessité l'utilisation d'un autre paquetage (`temp.expr.adaptation`) pour résoudre des problèmes de

compatibilité non traités par les opérateurs d'adaptations d'Hyper/J. Ils sont mis en évidence dans les lignes 20 à 30 du code source du paquetage qui est décrit ci-dessous.

```
32 package temp.expr.adaptation ;
33
34 abstract class ExprAffichage {
35   abstract String getType();
36
37   public void displayType() {
38      System.out.println(getType());
39   }
40 }
41
42 abstract class NumAffichage {
43   abstract int getValue();
44
45   public void displayValue() {
46      System.out.println(getValue());
47   }
48 }
49
50 abstract class OpBinaireAffichage {
51   abstract ExprAffichage getLeft();
52   abstract ExprAffichage getRight();
53
54   public ExprAffichage [] children() {
55      return new ExprAffichage [] {getLeft(),getRight()};
56   }
57 }
58
59 abstract class OpUnaireAffichage {
60   abstract ExprAffichage getOp();
61
62   public ExprAffichage [] children() {
63      return new Exp ExprAffichage[] {getOp()};
64   }
65 }
```

Ce nouveau paquetage permet de résoudre les problèmes de composition induits par l'affichage d'un arbre de syntaxe abstraite (AST). Il contient un ensemble de classes qui reproduit la même hiérarchie des classes que celle qui encapsule la syntaxe des expressions arithmétiques. Cet ensemble de classes permet de faire la transition avec les fonctionnalités nécessaires à l'affichage qui sont déjà présentes dans le paquetage `expr.syntaxe` mais sous une forme différente.

Les lignes 34 à 40 permettent par l'intermédiaire du module de composition d'ajouter à la classe `expr.syntaxe.Expr` la méthode `displayType()` qui est nécessaire à la classe `ASTNode` elle-même qui est fusionnée avec la classe `Expr`.

De même, les lignes 42 à 48 représentent une classe qui est composée dans la classe `expr.syntaxe.Num` pour lui permettre d'être compatible avec la classe `ASTleaf` (qui est composée dans `Num`). Les lignes 50 à 57 et les lignes 59 à 64 représentent respectivement les adaptations nécessaires des classes `expr.syntaxe.OpBinaire` et `expr.syntaxe.UpUnaire`.

Hyper/J produit un nouvel ensemble de classes encapsulé dans un nouveau sujet à partir des déclarations du module de composition. Ce nouveau sujet contient des classes synthétisées par la composition des trois préoccupations fonctionnelles.

Le mode de composition *ex-situ* d'Hyper/J qui est fortement fusionnel permet de réaliser la composition des trois préoccupations fonctionnelles sans modifier ces dernières.

Néanmoins cet exemple met en évidence les lacunes des opérateurs d'adaptation. Nous avons eu en effet recours à un ensemble de classes de transition pour composer convenablement les préoccupations. Cet ensemble de classes encapsule de manière fonctionnelle une partie de la composition. De plus, comme cela a été décrit dans la section précédente, Hyper/J ne permet pas

d'encapsuler des protocoles de composition, l'utilisateur doit donc spécifier, sans aide, les compositions.

Pour terminer il faut ajouter que la composition mise en œuvre dans cet exemple est difficile à comprendre car elle apparaît sous deux formes distinctes qui sont pourtant en interaction : le module de composition et l'ensemble des classes de transition.

3.2.5 Bilan sur les préoccupations fonctionnelles

Pour une application, le terme fonctionnel correspond à l'ensemble de ses fonctionnalités et concerne les algorithmes, la logique métier, ou encore le traitement de l'information. Les différentes préoccupations fonctionnelles d'une application doivent « travailler » en synergie pour réaliser les objectifs de celle-ci. Il est particulièrement important que les préoccupations fonctionnelles soient implémentées séparément. Cela permet à la fois, d'obtenir de meilleures capacités de réutilisation et de diminuer le couplage inter-préoccupations et par conséquent, la complexité des applications.

Dans cette section nous nous sommes en particulier posés les questions suivantes : ces préoccupations sont-elles séparables, et le cas échéant comment les séparer ? Est-il facile de réutiliser des préoccupations fonctionnelles au préalablement séparées ?

Dans la première partie nous avons présenté l'ensemble des solutions possibles proposées par l'approche orientée objets pour séparer et composer les préoccupations fonctionnelles. Aucune des solutions ne permet d'éviter à la fois la duplication de code ou le couplage entres les préoccupations fonctionnelles. Cette absence de solution de la part du paradigme objet rend impossible son utilisation pour séparer des préoccupations fonctionnelles.

La deuxième partie concerne la métaprogrammation et la seule solution qu'elle propose. Il s'agit d'utiliser la capacité générative de la méta programmation pour produire un nouvel ensemble de classes qui correspond à la composition des préoccupations fonctionnelles d'une application. La méta programmation permet donc de séparer les préoccupation fonctionnelles en les exprimant dans des ensembles de classes non couplées. Par contre, le fait de procéder par génération pour composer les préoccupations fonctionnelles *i)* ne permet pas la réutilisation de cette composition car elle dépend de l'ensemble des préoccupations à composer, *ii)* la composition est très difficile à implémenter car elle demande une excellente maîtrise du niveau méta et une parfaite connaissance des préoccupations en jeu. Ces différentes constatations nous amènent à considérer la métaprogrammation comme une approche non viable (surtout à cause de sa complexité).

La troisième partie montre que la programmation par aspects n'apporte aucune autre solution que celles apportées par l'objet car ce paradigme repose sur un mode de composition uniquement *in-situ*. Le seul avantage de l'approche et de bénéficier d'une séparation accrue par rapport à l'objet.

La quatrième partie présente les réponses apportées par la programmation par sujets. Ce paradigme permet de séparer les préoccupations fonctionnelles dans des classes différentes ; chaque préoccupation fonctionnelle est associée à un sujet constitué de toutes les classes qui s'y rapportent. Le paradigme des sujets utilise un mode de composition *ex-situ*, ce qui rend possible la composition d'un ensemble de préoccupations fonctionnelles ; le résultat obtenu est un nouvel ensemble de classes qui ne modifie pas les préoccupations composées. Par contre, le manque de capacité d'adaptation des opérateurs de composition oblige l'utilisateur à décrire des classes dites de transition pour composer convenablement les préoccupations. Ces classes encapsulent de manière fonctionnelle une partie de la composition. Elles rendent la réutilisation plus complexe car la composition est mise en œuvre à la fois par le module de composition et les classes de transition, qui interagissent entre eux, ce qui rend le processus de composition plus difficile à comprendre.

En conclusion, seule la programmation par sujets permet de séparer et de réutiliser des préoccupations fonctionnelles, mais elle ne fournit pas tous les outils qui permettraient de garantir une réutilisation facile.

3.3 LES PATRONS DE CONCEPTION

Les patrons de conception sont fréquemment utilisés dans le processus de construction du logiciel car ils apportent des solutions en matière de réutilisation. Cependant, leur implémentation dans le paradigme objet a tendance à être transversale à la hiérarchie de classes, et elle « pollue » ainsi le code correspondant.

Dans les sections précédentes nous nous sommes intéressés à l'intégration de préoccupations fonctionnelles et non fonctionnelles. Nous souhaitons maintenant appliquer la séparation des préoccupations aux patrons de conception. L'implémentation d'un patron de conception et ses diverses utilisations dans différents contextes doivent être clairement séparées pour éviter d'altérer leur lisibilité.

Cette section met en évidence la problématique associée à l'intégration des patrons de conception à travers l'étude de « l'observateur » [GHJ 99] que nous considérons suffisamment représentatif. En effet, son utilisation nécessite l'adaptation de ses clients d'un point de vue à la fois fonctionnel (en opérant sur les classes) et comportemental (en opérant sur les méthodes).

Chaque patron de conception correspond à un ensemble de classes qui décrit son protocole ; il représente une préoccupation de l'application au sens de la séparation des préoccupations [LH 95]. Typiquement un patron a vocation à être utilisé plusieurs fois dans l'application ; cette préoccupation doit donc dans un premier temps être séparée de ses clients, puis composée avec eux. La description de cette composition c'est-à-dire l'adaptation des clients, doit être séparée à la fois de celle des clients et de celle du patron.

Trois études [NK 01], [HB 02] et [HK 02] ont été menées autour de l'implémentation des patrons de conception avec la séparation des préoccupations (en utilisant AspectJ ou Hyper/J).

Dans la suite nous étudions la réutilisabilité du patron de conception « observateur » en nous appuyant à nouveau sur une comparaison des différentes approches (approches par objets, aspects et sujets). Avant de commencer cette comparaison, nous présentons l'exemple utilisé pour la comparaison.

3.3.1 Description de l'exemple

Le patron de conception observateur « définit une interdépendance de type *1 à plusieurs*, de façon telle que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour » [GHJ 99]. Sa mise en œuvre est décomposée en deux parties :

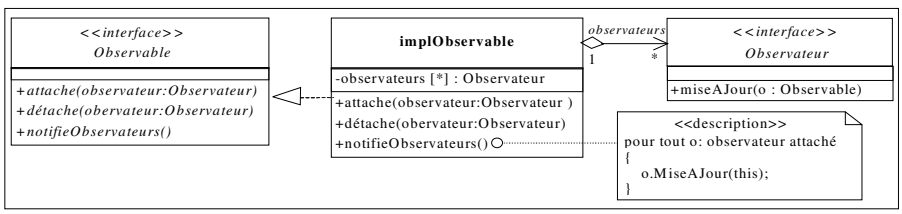


Figure 5, *Le patron de conception de l'observateur*

– Une partie composée des trois classes (voir figure 5) qui réifie les éléments du patron. Elle constitue son protocole : une entité *observable* permet à d'autres entités, les *observateurs*, de s'abonner pour recevoir des informations sur les modifications de son état. Cette partie est indépendante des classes qui vont réutiliser le patron.

– Une partie dépendante de l'utilisation du patron qui est formée de quatre adaptations. Trois sont fonctionnelles[1] et représentent l'état de l'objet observé, les liens d'héritage à modifier, et la méthode `miseAJour()` à implémenter dans les objets observateurs. La quatrième est comportementale[2] et correspond à l'ajout dans l'objet observable des appels à la méthode `notifieObservateurs()` qui prévient les observateurs de ses changements d'état.

Pour la suite de cette section nous garderons un même exemple d'utilisation : une interface homme machine qui consiste à encapsuler une interaction entre un bouton représenté par la classe `Button` et un label représenté par la classe `Label` qui l'observe. Ainsi quand on clique sur un bouton (appel à la méthode `Click()`) le label qui l'observe doit être modifié (appel à la méthode `colorCycle()`).

3.3.2 Patron de conception et programmation orientée objets

Pour rendre la classe `Button` observable dans un langage à héritage simple sans modifier son code, l'unique solution consiste à la sous-classer en créant une classe `ButtonObservable` qui implémente l'interface `Observable`. La classe `implObservable` décrite dans la figure 5 doit être dupliquée dans la classe `ButtonObservable` (lignes 6 à 8). Ceci devrait être exprimé (si c'était possible en Java) par un lien de réutilisation d'implémentation. La classe `Button-Observable` est présentée ci-dessous.

```
1   class ButtonObservable extends Button implements Observable {
2      public void click() {
3       super.click();
4       notifieObservateurs();
5      }
6     public void attache (Observateur o) { ... }
7     public void détache (Observateur o) { ... }
8     public void notifeObservateur ()    { ... }
9     private Observateur observateurs[] ;
10  }
```

Duplication de code depuis implObservable

Nous ne détaillerons pas la classe `LabelObservateur` qui s'implémente sur le même modèle, ni la mise en œuvre des deux classes pour l'exemple.

L'implémentation ci-dessus possède plusieurs inconvénients :

(1) *augmentation du nombre de classes et duplication de code* : l'obligation de créer une nouvelle classe `ButtonObservable` et d'y dupliquer une partie importante du patron de l'observateur (lignes 6 à 9),

(2) *modifications dans le reste de l'application* : les clients de la classe `Button` doivent dorénavant créer des objets de type `ButtonObservable`,

(3) *ajout difficile de nouvelles préoccupations* : pour ajouter de nouvelles préoccupations comme un autre patron, soit la classe `Button` doit être à nouveau sous-classée, soit c'est la classe `ButtonObservable` qui doit l'être. Le choix dépend du contexte d'utilisation des préoccupations ce qui oblige le programmeur à se reposer la question à chaque implémentation d'une préoccupation.

[1] L'adaptation fonctionnelle modifie les fonctionnalités d'une classe c'est-à-dire ses méthodes, ses variables, et son graphe d'héritage.
[2] L'adaptation comportementale modifie les fonctionnalités existantes d'une classe en modifiant ses méthodes par l'ajout de code avant, après ou autour d'elle.

Toutes les approches purement basées sur le paradigme de l'objet ne proposent aucune réponse pour ces trois inconvénients. On notera cependant que la présence de l'héritage multiple permet d'éviter la duplication de code signalé dans (1). De même, les classes génériques et les différents modèles de Mix-in bâties au dessus de la généricité [BC 90], [SB 98] et [Ost 02] ne permettent, comme l'héritage multiple, que d'éviter la duplication de code de (1). Toutes les approches citées ont une propriété commune : les classes sont des espaces de nom clos, ce qui sous-entend qu'aucune de ces approches ne permet de modifier des classes de l'extérieur.

La métaprogrammation permet de se libérer de cette contrainte, car les métaclasses peuvent modifier leurs classes associées. Pourtant, les inconvénients de la métaprogrammation précisés dans la section 3.2.2 s'appliquent aussi aux patrons de conception. L'unique solution offerte est d'utiliser la capacité générative de la métaprogrammation, or cette solution est dépendante de l'ensemble des préoccupations à appliquer à une classe et donc le travail fourni n'est pas réutilisable.

Quelle que soit l'approche choisie [OM 01] et même si on considère la métaprogrammation ou l'héritage multiple le paradigme de l'objet ne répond pas aux besoins [NK 01] [HB 02]. Les classes qui utilisent des patrons de conceptions sont difficilement réutilisables, car le code qui correspond au(x) patron(s) de conception est dispersé à travers toutes les classes clientes. Ceci engendre une pollution de ces dernières.

Le paradigme objet ne permet donc aucune séparation des préoccupations pour les patrons de conception. Nous suivons la même démarche que dans les sections précédentes et nous nous intéressons successivement aux solutions apportées par la programmation par aspects et la programmation par sujets.

3.3.3 Patron de conception et programmation par aspects

Comme dans les sections précédentes (où nous avons déjà présenté la programmation par aspects et AspectJ), nous allons utiliser le langage AspectJ [ASP 03] pour implémenter l'exemple.

L'implémentation de cet exemple nécessite l'utilisation de deux aspects comme le préconise [VWD 01]. L'objectif est ainsi d'améliorer la réutilisation des aspects et, par transitivité, celle de la préoccupation c'est-à-dire du patron de conception.

Un premier aspect abstrait représente le protocole de composition de la préoccupation. Un protocole de composition est le tissage abstrait et générique nécessaire à mettre en œuvre pour composer la préoccupation. Un deuxième aspect spécialise le premier par héritage [HU 01] ; il permet d'adapter la composition de la préoccupation à un contexte d'utilisation particulier.

Cette méthodologie est classique ; elle est en effet utilisée par les trois études [NK 01], [HB 02] et [HK 02] dont les objectifs étaient d'évaluer la capacité de réutilisation des patrons de conception lorsqu'ils sont mis en œuvre par séparation des préoccupations.

L'étude menée par [NK 01] sur les avantages de l'implémentation des patrons de conception avec AspectJ [KLM 97] ou Hyper/J [HO 93] ne répond pas à nos attentes en matière de la facilité de réutilisation, car l'implémentation des patrons de conception et leur composition sont encapsulées dans une même entité, ce qui diminue sa facilité de réutilisation et sa compréhension. Cette opinion est corroborée par celle de [HB 02] mais la solution qu'il propose n'est toujours pas, selon nous, satisfaisante, car il n'est pas tenu compte du fait qu'AspectJ ne peut pas encapsuler tout le protocole de composition d'un patron dans un aspect abstrait.

De même [HK 02] étudie aussi l'implémentation des patrons de conception en AspectJ avec des objectifs similaires qui sont une meilleure localisation du code, la facilité de réutilisation, et la facilité de composition. Leurs résultats sont meilleurs que [NK 01] et [HAC 01] pour les objectifs précédemment cités. Cela s'explique par une utilisation plus judicieuse du couple formé par

l'aspect abstrait et l'aspect concret suggéré dans [VWD 01] et [HU 01]. Celle-ci est la conséquence d'une meilleure maîtrise de la part de [HK 02] des possibilités de la programmation par aspects ce qui rend plus souple la méthode de [VWD 01] et [HU 01]. Les résultats de [HK 02] vont nous servir de référence.

La méthode employée par [HK 02] est similaire à [NK 01] sur un point : certains patrons de conception sont encapsulés directement dans les aspects sans utiliser de classes auxiliaires. [HB 02] a démontré que cela réduit la lisibilité et la compréhension du patron de conception et de l'aspect mais aussi diminue la facilité de réutilisation de l'aspect. Nous partageons l'opinion de [HB 02].

A propos de la délocalisation du code des préoccupations, nous pensons qu'utiliser la programmation orientée objets pour implémenter la préoccupation et un aspect abstrait pour implémenter le protocole de composition de la préoccupation, permet d'avoir une meilleure séparation entre la préoccupation et la manière de la composer.

Le choix de [HK 02] (pour certains patrons de conception, dont « observateur ») de mélanger la préoccupation et son protocole de composition dans un même aspect diminue la facilité de compréhension de la préoccupation. Ceci est dû en grande partie aux limites d'AspectJ qui sont mises en évidence par la suite dans la sous-partie 3.3.3.1 .

Le patron de conception est implémenté en Java par les classes de la figure 5. En parfaite cohérence avec l'approche proposée, l'aspect `ProtocoleObservateur` contient le protocole de composition :

```
1   import patrondeconceptions;
2   public abstract aspect ProtocoleObservateur {
3     abstract pointcut changementEtat(Observable s);
4     after(Observable s): changementEtat(s) {
5       s.notifieObservateurs();
6     }
7
8     private Vector Observable.observateurs=new Vector();
9     public void Observable.attache(Observateur obs) {
10      Observateurs.addElement(obs);
11    }
12    public void Observable.detache(Observateur obs) {
13      Observateurs.removeElement(obs);
14    }
15    public void Observable.notifieObservateurs() {
16      for (int i = 0; i < Observateurs.size(); i++)
17        ((Observateur)Observateurs.get(i)).miseAJour(this);
18    }
19  }
```

Adaptation comportementale

Adaptation fonctionnelle

La première partie de l'aspect (lignes 3 à 6) définit un point de jointure abstrait (ligne 3 : `changementEtat`) qui représente les changements d'état de l'objet « observé ». La méthode associée `notifieObservateurs` est exécutée après le point de jointure `changementEtat` (lignes 4 à 6) pour mettre à jour les « observateurs ».

Pour représenter complètement le protocole de composition l'aspect `ProtocoleObservateur` devrait ensuite déclarer deux autres adaptations fonctionnelles :

(1) Une première qui représente la classe qui doit devenir observable, cette adaptation est effectué en deux étapes :

(1.a) La classe à rendre observable implémente l'interface `Observable`,

(1.b) Celle-ci doit donc inclure l'implémentation des méthodes de la classe `implObservable` afin de l'empêcher de devenir abstraite.

(2) Adapter la classe qui observe pour qu'elle implémente l'interface `Observateur`.

Ces deux adaptations ne peuvent pas être déclarées dans l'aspect `ProtocoleObservateur` car AspectJ n'offre pas la possibilité d'utiliser des points de jointure abstraits pour réaliser des adaptations fonctionnelles.

Ces adaptations sont donc réalisées par l'aspect concret : (1.a) à la ligne 21, (2) à la ligne 25. Sauf pour (1.b) qui est réalisable dans l'aspect abstrait en concrétisant l'interface `Observable` (lignes 8 à 18).

La mise en œuvre de notre exemple est réalisée à travers l'aspect `Application_IHM` (lignes 20 à 29) qui hérite de `ProtocoleObservateur`. Il permet à un `Button` d'être observé (ligne 21) par un `Label` (ligne 25) qui change sa couleur (lignes 26 à 28, introduction de la méthode `miseAJour` dans la classe `Label`) quand le bouton est cliqué (lignes 22 à 23, concrétisation du point de jointure `changementEtat`).

```
20 public aspect Application_IHM extends ProtocoleObservateur {
21     declare parents: Button implements Observable;
22     pointcut changementEtat(Observable s):
23         target(s) && call(void Button.click());
24
25     declare parents: Label implements Observateur;
26     public void Label.miseAJour() {
27         colorCycle();
28     }
29 }
```

Rôle Observable

Rôle Observateur

Les quelques instructions qui suivent mettent en œuvre une association observateur - observable entre un `Button` (observable) et un `Label` (observateur) :

```
30 Button button= new Button (...)
31 Label label = new Label (...)
32 button.attache(label);
33 button.click();
```

La dernière ligne déclenche indirectement l'exécution de `notifieObservateur()` qui entraîne `label.MiseAJour()` et donc `label.ColorCycle()` grâce aux déclarations se trouvant dans l'aspect `Application_IHM`.

Le patron de conception est implémenté de manière modulaire sous forme de classes non couplées avec le reste de l'application. AspectJ a permis de le composer de manière non intrusive avec les classes qui doivent le réutiliser. Mais ceci n'a été possible que grâce à la transformation d'une partie de l'implémentation du patron de conception en aspect (voir lignes 8 à 18). Cette délocalisation d'implémentation diminue la lisibilité du patron de conception car elle empêche la complète séparation du patron de conception et de son protocole de composition.

Avant de conclure sur les avantages et les inconvénients d'AspectJ, nous allons étudier un autre exemple pour mettre en évidence certaines limites qui n'ont pas été complètement abordées jusqu'à présent, y compris les limites concernant la délocalisation dans les aspects de certains morceaux d'implémentation des préoccupations.

3.3.3.1 Limite de l'adaptabilité fonctionnelle d'AspectJ

Nous venons de voir une limite d'AspectJ avec l'étude du patron de conception de « l'observateur » : les possibilités d'adaptabilité fonctionnelle. Ces limites ont été contournées grâce à la méthode proposée par [HK 02] ce qui a permis d'implémenter l'exemple. Néanmoins, nous allons voir que cette méthode ne s'applique pas avec succès dans tous les cas. Pour montrer les limites de cette approche nous allons étudier l'implémentation qu'ils proposent pour le patron de conception « composite » ci-dessous.

Le patron de conception composite permet de voir uniformément (interface `Component`) des objets qu'ils soient composites (interface `Composite`) ou simples (interface `Leaf`).

```
1   public abstract aspect CompositionProtocol {
2
3     protected interface Component {};
4     protected interface Composite {} extends Component;
5     protected interface Leaf      {} extends Component;
6
7     private WeakHashMap perComponentChildren = new WeakHashMap();
8
9     private Vector getChildren(Component s) {
10      Vector children;
11      children = (Vector) perComponentChildren.get(s);
12      if (children == null ) {
13        children = new Vector();
14        perComponentChildren.put(s, children);
15      }
16      return children;
17    }
18
19    public void addChild(Composite composite,
20                         Component component) {
21      getChildren(composite).add(component);
22    }
23
24    public void removeChild(Composite composite,
25                            Component component) {
26      getChildren(composite).remove(component);
27    }
28
29    public Enumeration getAllChildren(Component c) {
30      return getChildren(c).elements();
31    }
32
33    protected interface FonctionVisitor {
34      public Object doIt(Component c);
35    }
36
37    protected static Enumeration recurseFonction(Component c,
38                                                 FonctionVisitor fv){
39      Vector results = new Vector();
40      for (Enumeration enum = getAllChildren(c);
41           enum.hasMoreElements() ; ) {
42        Component child;
43        child = (Component) enum.nextElement();
44        results.add(fv.doIt(child));
45      }
46      return results.elements();
47    }
48  }
```

Cet aspect abstrait encapsule le patron de conception. Il définit quatre interfaces (lignes 3 à 5 et lignes 33 à 35) qui représentent les entités de l'objet composite. Il implémente aussi les méthodes (`addChild`, `removeChild`, …) pour manipuler ses entités, c'est-à-dire ajouter ou enlever des composants à un objet composite, et pour exécuter une méthode sur l'ensemble des objets appartenant à un composant.

La méthode d'implémentation permet de cacher les fonctionnalités partagées entre toutes les classes qui sont clientes du patron de conception composite. Les fonctionnalités sont présentes uniquement dans l'aspect. Pour y accéder à partir d'une instance d'une classe cliente il faut interroger l'aspect associé à cette instance par l'intermédiaire de la méthode `AspectOf()(`qui est ajoutée à toute classe qui subit une adaptation par un aspect). Une fois l'instance de l'aspect récupérée on peut ensuite utiliser les méthodes du patron qui sont déclarées dans l'aspect (comme par exemple `addChild`, `replaceChild`, etc.). Le code ressemble alors à `monInstance.AspectOf().addChild()` (modulo quelque transtypages).

Comme nous l'avons fait remarquer dans l'étude de l'implémentation du patron de conception « observateur », la concrétisation des interfaces ou la délocalisation des méthodes qui se rapportent aux préoccupations dans les aspects proviennent toutes deux d'un manque d'adaptabilité fonctionnelle de la programmation par aspects (ceci est donc valable en particulier pour AspectJ).

Il manque donc à la programmation par aspects un mécanisme d'adaptation fonctionnelle comme c'est le cas par exemple dans la programmation par sujets. Il permettrait d'abstraire un ensemble d'adaptations fonctionnelles pour un ensemble de classes. Son absence oblige le programmeur à implémenter partiellement des protocoles de composition, incomplètement décrits dans des aspects abstraits.

L'alternative proposée par [HK 02] oblige les classes clientes à connaître le contenu de l'aspect pour utiliser les fonctionnalités communes du patron de conception. Ceci ne favorise pas la compréhension, car même si les aspects modifient des classes pour y ajouter de nouvelles préoccupations, certaines fonctionnalités des préoccupations restent dans les aspects (toujours à cause du manque d'expressivité du mécanisme d'adaptation fonctionnelle).

La méthode utilisée par [HK 02] contraint donc la classe cliente d'une préoccupation à utiliser à son tour l'aspect qui la modifie pour accéder à la méthode induite (par l'aspect) `AspectOf(Object o)`. Elle pourra ainsi avoir accès aux fonctionnalités offertes par le patron (ex : `addChild`) à ses classes clientes.

Cette indirection diminue selon nous la compréhension et la lisibilité de l'application car elle nécessite de la part des utilisateurs des classes clientes du patron de conception une connaissance précise de son implémentation. Nous pensons que les utilisateurs ne doivent pas avoir à se soucier de ce genre de détail.

Nous étudions maintenant la spécialisation de l'aspect ci-dessous pour mettre en évidence de nouveaux problèmes.

```
49 public aspect FileSystemComposite extends CompositeProtocol {
50
51   declare parents:  Directory implements Composite;
52   declare parents:  File      implements Leaf;
53
54   public int sizeOnDisk(Component c) {
55     return c.sizeOnDisk();
56   }
57
58   private abstract int Component.sizeOnDisk();
59
60   private int Directory.sizeOnDisk() {
61     int diskSize = 0;
62     java.Util.Enumeration enum;
63     for(enum = this.aspectOf().getAllChildren(this) ;
64         enum.hasMoreElement() ; ) {
65       diskSize += ((Component)enum.nextElement()).sizeOnDisk();
66     }
67     return diskSize;
68   }
69
70   private int File.sizeOnDisk() {
71     return size;
72   }
73 }
```

Cet aspect montre une réutilisation de la composition du patron de conception dans un contexte nouveau, celui de deux classes `Directory` et `File`. Le patron de conception permet d'ajouter les fonctionnalités nécessaires (accessibles uniquement par l'aspect à cause des limites d'adaptabilité fonctionnelles), pour accéder aux deux classes de manière homogène, en utilisant l'interface `CompositionProtocol.Composite` (voir lignes 51 à 52) pour représenter une arborescence de fichiers. De même le patron de conception a été utilisé pour ajouter une nouvelle fonctionnalité (lignes 54 à 72) afin de pouvoir calculer la taille prise sur un disque par un `Component`.

L'implémentation proposée par [HK 02] possède un autre défaut qui ne lui permet d'être utilisée qu'une seule fois dans l'application. En effet, si le patron de conception est utilisé plusieurs fois dans la même application et que ces différentes utilisations modifient l'interface `CompositionProtocol.Composite` pour y ajouter des fonctionnalités (et donc bénéficier du patron pour avoir un accès plus homogène à ces fonctionnalités), alors la composition est incorrecte.

Cette erreur de composition provient du fait que tous les aspects concrets (ils représentent l'ensemble des réutilisations d'une même préoccupation) qui héritent d'un même aspect abstrait peuvent modifier le même ensemble d'interface. En d'autres termes, si des aspects concrets modifient les interfaces qui sont présentes dans l'aspect abstrait pour y ajouter de nouvelles fonctionnalitées (dans notre exemple c'est l'interface `CompositionProtocol.Composite`), alors toutes ces fonctionnalités seront communes à l'ensemble de ces aspects.

Ce dernier point pose un réel problème car les interfaces définies dans l'aspect abstrait sont la plupart du temps utilisées pour ajouter leurs méthodes dans les classes clientes de la préoccupation. Ainsi l'ensemble des classes clientes d'un même patron de conception partage toutes les fonctionnalités ajoutées par tous les aspects concrets.

Dans notre exemple l'aspect concret utilise les interfaces `CompositionProtocol.Composite` et `CompositionProtocol.Leaf` pour ajouter des méthodes dans les classes clientes `Directory` et `File`. Toute nouvelle utilisation du patron de conception bénéficiera des ajouts réalisés par l'aspect concret `FileSystemComposite` (et inversement).

Cette erreur de composition provient du partage des interfaces (qui sont définies dans l'aspect abstrait) entre tous les aspects concrets (dans notre exemple l'interface `CompositionProtocol.Composite` et ses sous interfaces) et donc entre tous les clients de la préoccupation.

Cette erreur de composition comme cela a déjà été dit, provient du partage des interfaces. Ce partage résulte lui-même de la nature *in-situ* [OK 00] du mécanisme adaptation offert par le modèle de composition de la programmation par aspects.

[HK 02] a proposé des solutions meilleures que [NK 01] et [HB 02] ; pourtant ces solutions souffrent de défauts inhérents au modèle des aspects.

3.3.3.2 Bilan

L'adaptabilité du patron de conception est restreinte par le manque de possibilités proposées par AspectJ pour réaliser des adaptations fonctionnelles (voir section 3.2.3). En effet, il n'est pas possible de définir dans un aspect un lien de « réutilisation d'implémentation » qui permettrait d'introduire dans une classe l'implémentation de méthodes fournies par d'autres classes. Pour contourner ce problème, AspectJ met en œuvre des adaptations d'interfaces. Or, comme la composition fournie par le modèle de la programmation par aspects est *in-situ* [OK 00], nous nous retrouvons confrontés au problème de la fragilité des classes de base [MS 98]. Ces limitations ont un impact direct sur la facilité de réutilisation et les possibilités d'adaptation.

Malgré la possibilité offerte par AspectJ d'utiliser un couple aspect abstrait / aspect spécialisé, la facilité de réutilisation est amoindrie [DEV 01] par l'impossibilité d'encapsuler tout le protocole de composition dans l'aspect abstrait. L'utilisateur doit donc compenser dans l'aspect spécialisé l'absence de ce service, sans contrôle sur la composition de la part d'AspectJ.

Dans la suite, nous abordons le même exemple en utilisant la programmation par sujets.

3.3.4 Patron de conception et programmation par sujets

Revenons à l'exemple proposé dans la section 3.3.1. Il contient deux préoccupations qui sont représentées par deux sujets. Le patron de conception de « l'observateur » est encapsulé par identification des classes qui le compose dans un premier sujet (HyperSlice) nommé `PatronDeConceptions.Observateur` (lignes 2 à 3). Ce sujet contient les trois classes implémentées en Java de la figure 4 : `Observable`, `Observateur` et `implObservable`. Un deuxième sujet `Application.IHM` (ligne 5) contient les classes `Button` et `Label` qui vont être composées avec les classes du patron.

Hyper/J [HYP 03] ne permet pas d'encapsuler des protocoles de composition, car les modules de composition ne possèdent ni l'héritage, ni l'abstraction. Les deux sujets précédents sont donc directement composées pour produire un nouveau sujet : `Application.IHMetendue`.

Le module de composition `Application.IHMetendue` (lignes 6 à 19) contient la composition du patron de conception (ligne 8) et des classes qui doivent l'utiliser (ligne 7). Le sujet composite `Application.IHMetendue` (lignes 5 à 19) représente la réutilisation du patron de conception : la classe `Button` devient `Observable` (ligne 12), la classe `Label` devient un `Observateur` (ligne 16), la méthode click de la classe `Button` provoque une mise à jour des observateurs (lignes 13 à 14) et l'appel de la méthode `MiseAJour()` sur un `Label` déclenche l'exécution de la méthode `colorCycle()` (lignes 17 à 18).

```
1   -concerns
2      package patrondeconceptions : PatronDeConceptions.Observateur;
3      package application.ihm     : Application.IHM ;
4   -hypermodules
5     hypermodule Application.IHMetendue
6       hyperslices:
7         Application.IHM,
8         PatronDeConceptions.Observateur;
9       relationships:
10        mergebyname;
11
12        equate class Button , Observateur.implObservable;
13        bracket Button.Click with after
14          Observateur.implObservable.notifieObservateurs();
15
16        equate class Label  , Observateur.Observateur;
17        bracket Observateur.MiseAJour with after
18          Label.colorCycle();
19    end hypermodule;
```

Hyper/J a permis d'implémenter de manière modulaire le patron de conception sous forme de classes Java non couplées avec le reste de l'application puis de le composer de manière non intrusive avec les classes qui doivent l'utiliser grâce à un module de composition.

Pourtant, la réutilisation du patron de conception de l'observateur est difficile car Hyper/J ne permet pas d'encapsuler un protocole de composition. L'utilisateur du patron de conception doit donc respecter le protocole de composition de l'observateur sans contrôle de la part d'Hyper/J.

3.3.5 Bilan sur les patrons de conception

Cette section a présenté une étude relative à la réutilisation des patrons de conception. Bien que ces derniers soient fréquemment utilisés, leur implémentation à une tendance à être transversale à la hiérarchie de classes et « pollue » ainsi le code des classes qui la compose. De plus, les patrons de conception sont une sorte de préoccupation hybride entre fonctionnel et non fonctionnel. En effet, ils apportent des services à l'application, ces services sont souvent de niveau fonctionnel ; c'est le cas du patron « observateur ».

Cette section apporte des réponses dans les domaines suivants : capacité des patrons de conception à être séparés, méthodologies de séparation, et enfin facilité de réutilisation d'une implémentation de patron de conception qui a été séparée au préalable.

La première partie montre que le paradigme de l'objet (avec ou sans métaprogrammation et avec ou sans généricité) ne permet pas de séparer l'implémentation des patrons de conception de leur classe cliente.

La deuxième partie expose le fait que la programmation par aspects permet de séparer les patrons de conception en encapsulant ces derniers à deux endroits différents : dans des classes et dans l'aspect abstrait qui encapsule de protocole de composition. Néanmoins, la réutilisation des patrons de conception est difficile car la programmation par aspects ne permet pas d'encapsuler de manière abstraite et complète les protocoles de composition. L'utilisateur doit donc compenser cette lacune dans l'aspect qui spécialise le protocole de composition abstrait.

La troisième partie met en évidence le fait que la programmation par sujets, même si elle permet de séparer et composer les patrons de conception, n'apporte pas non plus une solution pleinement satisfaisante. En effet, la réutilisation des patrons de conception est difficile car Hyper/J ne permet pas d'encapsuler un protocole de composition. L'utilisateur du patron de conception doit donc compenser cette limitation sans aide du langage.

En conclusion, le paradigme objet n'apporte aucune solution au problème de séparation des patrons de conception tandis que la programmation par aspects ou la programmation par sujets ap-

portent toutes deux des solutions. Cependant on constate que ce sont les défauts liés à l'encapsulation des protocoles de composition qui rendent toutes les solutions difficilement utilisables.

3.4 SYNTHESE DES RESULTATS

L'étude menée dans les sections précédentes a montré les apports et les limitations du paradigme objet et de certaines de ses extensions (approches par aspects et par sujets) concernant la prise en compte de trois catégories de préoccupation. Nous allons maintenant nous appuyer dessus pour répondre à la question suivante : « Quels sont les moyens nécessaires pour séparer et composer les préoccupations dans le paradigme de l'objet en ayant pour objectif de faciliter la réutilisation des préoccupations ? »

Un premier enseignement que nous avons tiré de cette étude est que même si on réussit à séparer les préoccupations en enlevant leur inter-couplage, la principale difficulté reste la mise en œuvre de leur composition pour produire l'application escomptée.

Reformer l'application à partir de préoccupations séparées est une opération qui peut être faite soit avant la compilation soit pendant l'exécution[1]. Cependant, la complexité des préoccupations empêche que cette opération soit entièrement automatisée. On constate par ailleurs, en étudiant divers langages (AspectJ et Hyper/J en particulier), qu'il est possible d'implémenter un tisseur. Celui-ci, à partir de l'ensemble des préoccupations et d'un plan de composition permet de produire l'application finale par composition.

Ce plan de composition décrit toutes les adaptations à mettre en oeuvre pour composer les diverses préoccupations de l'application. C'est dans ce plan que réside la principale difficulté.

Une première simplification consiste à découper ce plan de composition en éléments plus petits qui ne portent que sur des sous-ensembles de préoccupations à composer ensemble. L'approche suivie par la programmation par sujets pour composer des préoccupations se calque exactement sur cette méthodologie : chaque composition produit des sujets composites qui peuvent être à leurs tour composés avec d'autres sujets.

Comme l'ont aussi montré [VWD 01] et [HU 01], la composition d'une préoccupation est facilitée si cette dernière est accompagnée de son protocole de composition.

Le protocole de composition d'une préoccupation est l'abstraction de l'ensemble des adaptations qu'il est nécessaire de réaliser pour pouvoir composer la préoccupation avec ses clients dans l'application. Le protocole de composition définit une « bonne » façon de composer la préoccupation.

Comme nous travaillons dans le paradigme objet, un protocole de composition doit pouvoir être réifié dans ce paradigme. Il correspond à une entité abstraite qui doit être spécialisée et concrétisée par héritage. La spécialisation d'un protocole de composition permet d'adapter ce dernier hors des limites qu'il a fixé *a priori* (ceci n'est pas anormal, il est en effet difficile de prévoir tous les cas de figure).

La concrétisation d'un protocole de composition doit réaliser effectivement la composition mais aussi proposer une solution (adaptée à la configuration des clients) pour compléter les définitions qui le nécessitent dans le protocole (ce dernier étant abstrait).

[1] Il est clair que dans ce cadre de nombreux problèmes supplémentaires interviennent et la mise en œuvre en devient encore plus délicate.

Les trois sections précédentes mettent en particulier les conclusions suivantes en évidence (résumées dans le tableau 2). Elles concernent les capacités d'encapsulation et d'abstraction des protocoles de composition.

Paradigme/Modèle	Encapsulation	Abstraction
Objet	Non	Non
AspectJ	Incomplète	Incomplète
Hyper/J	Non	Non

Tableau 2. *Encapsulation et abstraction des protocoles de composition dans les approches étudiées*

Seule l'approche par aspects (avec AspectJ en particulier) permet d'encapsuler et d'abstraire *une partie* seulement des protocoles de composition. Ceci provient du fait qu'un protocole de composition est un ensemble d'adaptations et qu'AspectJ (et le modèle de la programmation par aspect en général) ne peut pas traiter tous les types d'adaptation. De plus, comme le montre [EL 03], AspectJ possède certaines limites dans le polymorphisme entre aspects qui réduit la réutilisation d'aspect par héritage.

Pour présenter plus en détail les capacités d'encapsulation et d'abstraction des protocoles de composition, il est utile d'évoquer les capacités d'adaptation des différents modèles. Pour cela nous nous appuyons sur tous les types d'adaptation qui ont été nécessaires pour réaliser la composition des préoccupations non fonctionnelles, des préoccupations fonctionnelles, et des patrons de conception (préoccupations hybrides). La liste complète de ces adaptations est présentée dans le tableau 3 ci-dessous :

Description	Type d'adaptation	Cible de l'adaptation
Implémenter de nouvelles interfaces	Fonctionnelle	Classe, ensemble de classes
Fusion de classes et de méthodes	Fonctionnelle	Classe, ensemble de classee
Ajouter ou redéfinir des méthodes dans des classes	Fonctionnelle	Classe, ensemble de classes
Ajouter de nouvelles variables d'instance (et de classe) à des classes	Fonctionnelle	Classe, ensemble de classes
Interception avant, après, autour, sur exception des méthodes	Comportementale	Méthodes, ensemble de méthodes
Interception des accès aux variables d'instance (et de classe)	Comportementale	Une variable, ensemble de variables

Tableau 3. *Ensemble des adaptations nécessaires pour composer des préoccupations dans le paradigme de l'objet*

L'étude que nous avons conduite suggère de classer les adaptations en deux catégories :

– Les adaptations fonctionnelles. Elles modifient les fonctionnalités d'une classe : ses méthodes, ses variables, et son graphe d'héritage.

– Les adaptations comportementales. Elles modifient les fonctionnalités existantes d'une classe en modifiant ses méthodes par ajout de code avant, après ou autour d'elle.

Cette classification, nous a permis, au vu des résultats (voir tableau 4) relatifs aux capacités d'adaptations des approches étudiées, de classifier la programmation par aspects comme étant plus particulièrement dédiée à l'adaptation comportementale et la programmation par sujets comme étant spécialisée dans l'adaptation fonctionnelle.

Description	Objet	AspectJ	Hyper/J
Implémenter de nouvelles interfaces	Non	Oui	Oui
Fusion de classes et de méthodes	Non	Non	Oui
Ajouter ou redéfinir des méthodes	Non	Oui	Oui
Ajouter de nouvelles variables d'instance (et de classe)	Non	Oui	Oui
Interception avant, après, autour, sur exception des méthodes	Non	Oui	Partiellement
Interception des accès aux variables d'instance (et de classe)	Non	Oui	Non

Tableau 4. *Support des différents types d'adaptation pour les approches étudiées.*

Tous les résultats synthétisés dans le tableau 4 montrent que le paradigme de l'objet seul ne permet pas de composer les préoccupations. Nous avons donc choisi de ne plus le mentionner dans les prochains tableaux de synthèse. Ces résultats montrent aussi qu'AspectJ et Hyper/J ne permettent pas de réaliser tous les types d'adaptions nécessaires.

L'étude de l'ensemble de ces résultats met en évidence les raisons pour lesquelles AspectJ ne propose qu'un support incomplet de l'encapsulation des protocoles de composition (voir tableau 2). En effet, comme la programmation par aspects n'offre pas l'expressivité suffisante pour réaliser la fusion de classes ou de méthodes (voir tableau 4), elle ne peut pas non plus encapsuler complètement un protocole de composition lorsqu'il nécessite ce type d'adaptation.

L'étude que nous avons menée dans les sections précédentes permet aussi de montrer les capacités d'abstraction des adaptations des approches étudiées. Elles sont résumées dans le tableau 5 ci-contre.

Description	AspectJ	Hyper/J
Implémenter de nouvelles interfaces	Non	Non
Fusion de classes et de méthodes	Non	
Ajouter ou redéfinir des méthodes	Non	
Ajouter de nouvelles variables d'instance (et de classe)	Non	
Interception avant, après, autour, sur exception des méthodes	Oui	
Interception des accès aux variables d'instances (et de classe)	Oui	

Tableau 5. *Capacité d'abstraction des adaptations d'AspectJ et d'Hyper/J.*

Ces résultats (voir tableau 5) rappellent qu'Hyper/J ne possède aucune relation d'héritage sur ses modules de composition. Ces derniers ne peuvent donc pas disposer de capacités d'abstraction pour encapsuler des protocoles de composition.

AspectJ (et la programmation par aspects de manière plus générale) est conçu plus particulièrement pour prendre en compte des adaptations comportementales et beaucoup moins pour réaliser des adaptations fonctionnelles qu'il ne sait pas abstraire. Ceci permet d'expliquer pourquoi AspectJ ne permet de traiter efficacement qu'une partie seulement des protocoles de composition (voir tableau 2).

En complément il est intéressant de noter que, pour pouvoir abstraire des adaptations il faut en particulier être capable d'abstraire leurs cibles (les entités concernées). Or, comme le montrent les résultats qui sont présentés dans le tableau 6, ni AspectJ, ni Hyper/J ne permettent d'abstraire tous les types de cible possibles.

Description	AspectJ	Hyper/J
Classe ou ensemble de classes	Non	Non
Méthode ou ensemble de méthodes	Oui	Non
Variable d'instance ou ensemble de variables d'instances	Non	Non

Tableau 6. *Possibilités d'abstraction des cibles des adaptations***.**

Après avoir étudié le support des protocoles de composition dans les différentes approches, nous nous intéressons de manière plus approfondie aux capacités d'adaptation des approches étudiées.

Les adaptions portent sur des cibles différentes, comme présenté dans la troisième colonne du tableau 3. Cette étude permet de montrer (voir le tableau 7) que la programmation par aspects et la programmation par sujets ne permettent pas de décrire toutes les cibles possibles.

Description	AspectJ	Hyper/J
Une classe	Oui	Oui
Un ensemble de classes	Oui	Oui
Un ensemble de classes représenté par des expressions régulières	Non	Non
Une méthode (avec référence à la classe associée)	Oui	Oui
Un ensemble de méthodes d'une même classe	Oui	Oui
Un ensemble de méthodes sur des classes différentes avec des expressions régulières.	Oui	Non

Tableau 7. *Support des différentes cibles necessaires aux adaptations par AspectJ et Hyper/J.*

La composition d'une préoccupation peut être de deux types : in-situ ou ex-situ. Le tableau 8 montre le support de ces types de composition par AspectJ et Hyper/J.

Description	AspectJ	Hyper/J
Composition In-Situ	Oui	Non
Composition Ex-Situ	Non	Oui

Tableau 8. *Support des différents types de composition.*

A partir de l'ensemble de ces résultats nous proposons dans la section suivante, un bilan sur le support de la séparation et de la composition des préoccupations dans le paradigme de l'objet.

3.5 BILAN

Cette étude a permis de dénombrer dans un premier temps les différents types de préoccupations qui sont les plus susceptibles d'être utilisés lors de la conception et à l'implémentation d'applications.

Un exemple a permis pour chacun des types de préoccupation étudiés de mettre en évidence les apports et limitations des différents paradigmes de programmation : *objets*, *métaprogrammation*, *aspects*, et *sujets*. La facilité de réutilisation et la capacité d'adaptation des préoccupations ont servi de base à la comparaison des différents paradigmes. Nous avons en particulier utilisé ces deux propriétés pour mesurer leur capacité de composition.

Nous proposons dans un premier temps de récapituler brièvement les avantages et les défauts rencontrés avec chacun des paradigmes. Ceci va permettre de définir l'ensemble des opérateurs de composition nécessaires pour une séparation et une composition des préoccupations dans le paradigme de l'objet.

L'étude de la prise en compte des préoccupations non fonctionnelles (voir section 3.1) a permis de mettre en évidence plusieurs limitations. Le paradigme objet ne permet pas de les encapsuler, ce qui conduit à de la duplication de code et à la modification des classes clientes. La métaprogrammation permet de les séparer et de les composer avec leurs clients mais la complexité inhérente à l'approche nécessite des programmeurs hautement qualifiés. La programmation par aspects a permis de séparer et de composer la préoccupation en conservant la facilité de réutilisation et la capacité d'adaptation. La programmation par sujets a permis de séparer et de composer la préoccupation mais sa réutilisation est difficile et les capacités d'adaptation minimales.

La même étude appliquée aux préoccupations fonctionnelles (voir section 3.2) montre qu'il a été difficile de les composer. La programmation orientée objets, la métaprogrammation, et la programmation par aspects n'ont apporté aucune solution viable. Seule la programmation par sujets a permis de composer et de séparer les préoccupations mais, une nouvelle fois, la réutilisation est peu aisée et il n'y a aucune capacité d'adaptation.

L'étude sur la facilité de réutilisation d'un patron de conception menée dans la section 3.3 a permis de mettre en évidence de nombreuses lacunes dans les trois approches évaluées (objet, aspect et sujet). La programmation orientée objets et la métaprogrammation n'ont pas permis de découpler la préoccupation de ses clients. La programmation par aspects a permis de séparer et de composer le patron de conception mais avec : *i)* de la duplication de code, *ii)* une facilité de réutilisation moyenne et *iii)* peu de possibilités d'adaptation. La programmation par sujets a permis de composer et de séparer la préoccupation mais sans pouvoir bénéficier d'une quelconque facilité de réutilisation ou d'adaptation.

Nous pouvons déduire de cette étude que les différents modèles bâtis au dessus du paradigme de l'objet ne permettent pas d'avoir les deux propriétés qui nous semblent importantes : la facilité de réutilisation et la capacité d'adaptation. Néanmoins cette étude permet de mettre en évidence les pré-requis.

Nous allons maintenant nous intéresser à la liste des opérateurs d'adaptation qui permettent de réutiliser efficacement les préoccupations dans le paradigme de l'objet.

Les exemples relatifs aux patrons de conception et aux préoccupations non fonctionnelles mettent en évidence la nécessité d'avoir deux modes de composition : *in-situ* (adaptation directe des classes) et *ex-situ* (création de nouvelles classes qui représentent l'adaptation). Ces deux modes de composition doivent être accessibles grâce aux opérateurs d'adaptation.

Les exemples concernant les patrons de conception et les préoccupations fonctionnelles nécessitent l'encapsulation d'un protocole de composition. Ce protocole doit pouvoir être spécialisé pour faciliter, assister et permettre sa réutilisation. Il est donc nécessaire que les opérateurs de composition prennent en charge l'abstraction et la spécialisation. Ceux qui sont dédiés à l'abstraction permettent d'implémenter le protocole de composition et ceux qui concernent la spécialisation permettent d'adapter et de réaliser le protocole pour un contexte de réutilisation donné.

Tous les opérateurs d'adaptation doivent pouvoir s'appliquer sur des points de jointure[1]. Il doit être possible d'en définir qui soient abstraits ou concrets. C'est essentiel pour répondre au problème d'encapsulation des protocoles de composition. De plus, leur granularité doit être variable et aller de l'ensemble de classes jusqu'à la méthode.

Comme les cibles des points de jointures peuvent être des éléments de type différents, les points de jointure doivent posséder un type (par exemple : ensemble de méthodes, ensemble de classes). L'utilisation du typage permet d'éviter les erreurs d'association entre points de jointure et opérateurs d'adaptation, de faciliter la spécialisation et de rendre les protocoles de composition plus sûrs.

[1] Ce sont des éléments de la sémantique du langage sur lesquels les opérateurs d'adaptation peuvent agir par adaptation.

Les points de jointure doivent aussi pouvoir transporter du contexte en provenance de leurs cibles. On peut citer par exemple les arguments des méthodes, l'instance courante, la classe ou la méthode appelante ou appelée. Ce contexte doit être utilisable par les opérateurs d'adaptation.

Les points de jointure doivent pouvoir être associés à un texte libre les décrivant, ceci permet de faire des opérateurs d'adaptations une partie complète de la documentations des préoccupations. Ce texte pourra ensuite être utilisé par exemple par le compilateur comme aide mémoire si jamais l'utilisateur a éventuellement oublié de spécialiser un point de jointure abstrait.

Les constatations précédentes montrent que les opérateurs d'adaptation suivants sont nécessaires pour réutiliser facilement les préoccupations dans le paradigme de l'objet :

– Adaptation d'une classe pour implémenter de nouvelles interfaces
– Fusion de classes et de méthodes
– Ajout ou redéfinition des méthodes dans des classes
– Ajout de nouvelles variables d'instance (et de classe) à des classes
– Interception avant, après, autour ou sur exception, des méthodes
– Interception des accès aux variables d'instance (et de classe)

Cet ensemble d'opérateurs conduit à l'élaboration d'un nouveau modèle dont le cahier des charges est présenté dans le chapitre 4. Ce modèle a pour objectif de favoriser la réutilisation et il s'appuie sur les limitations relevées dans le cadre de cette étude. Il sera présenté dans le chapitre 5. Sa mise en œuvre sera ensuite décrite dans le chapitre 6. Cette mise en œuvre va permettre de confronter notre modèle aux mêmes exemples que ceux que nous avons utilisé dans l'étude ci-dessus ; nous espérons ainsi montrer les améliorations apportées par notre approche dans le chapitre 7.

Chapitre 4

Cahier des charges du modèle

Ce chapitre présente le cahier des charges que doit respecter un modèle dédié à l'amélioration de la réutilisation des préoccupations. Nous allons dans un premier temps exposer les motivations et les buts de ce nouveau modèle face aux approches existantes. Dans un second temps, nous détaillons les différentes fonctionnalités que doit supporter le modèle.

4.1 MOTIVATIONS

L'étude sur la réutilisation des préoccupations du chapitre 3 met en évidence les limitations des langages de programmation avec séparation des préoccupations définis au-dessus de paradigme de l'objet. Parmi tous les langages disponibles (voir section 2.4), seul AspectJ [ASP 03] et Hyper/J [HYP 03] ont été sélectionnés. Ces deux langages sont donc les approches les plus abouties pour faire de la séparation des préoccupations dans le paradigme de l'objet.

L'étude approfondie du chapitre 3 analyse la qualité de la séparation et de la composition de trois types de préoccupations : non fonctionnelle, fonctionnelle et une combinaison des deux. Ces résultats peuvent se généraliser car les trois types de préoccupations étudiées correspondent aux besoins pour le paradigme de l'objet en matière de séparation des préoccupations.

Les résultats de cette étude peuvent être classifiés selon quatre catégories :

– La capacité à séparer une préoccupation du reste de l'application ;
– La complétude de l'encapsulation du protocole de composition de la préoccupation ;
– La facilité de réutilisation d'une préoccupation dans différents contextes ;
– Les possibilités d'adaptation d'une préoccupation.

Aucune des trois grandes familles de type de préoccupation n'a pu être séparée et composée de manière optimale (voir 3.4). Le paradigme objet et les différentes approches étudiées présentent donc de réelles limitations relativement à la séparation des préoccupations.

Au cours de l'étude du chapitre 3 nous avons présenté des premiers éléments de solution pour répondre à ces limitations. Nous désirons ici définir le cadre d'un nouveau modèle hybride entre la programmation par aspects [KLM 97] et la programmation par sujets [HO 93]. Le choix de ces deux approches repose sur les résultats des études présentées dans les chapitres 2 et 3 qui ont montré que ces deux modèles sont les plus élaborés et qu'il permettrait ensemble de résoudre la majeure partie des problèmes constatés par le chapitre 3.

Dans la section suivante nous présentons les différentes qualités que doit avoir le modèle en précisant les similitudes avec les aspects ou les sujets mais aussi ses spécificités pour un support complet de la séparation des préoccupations dans le paradigme de l'objet.

4.2 ARCHITECTURE DU MODELE

Cette partie aborde les différentes facettes que devra comporter le modèle. Nous allons voir dans un premier temps sous quelle forme le modèle doit encapsuler des préoccupations. Dans un second temps nous abordons l'encapsulation de la composition des préoccupations et ses différents modes de composition. Dans un troisième temps nous évoquons l'expressivité des opérateurs d'adaptation.

4.2.1 Modélisation des préoccupations

Les langages supportant la séparation des préoccupations (voir section 2.4) utilisent tous les classes pour encapsuler les différentes préoccupations. Une préoccupation est par contre toujours isolée dans la notion d'ensemble de classes (comme par exemple la notion de *package* en Java ou en UML).

Un seul modèle propose une nouvelle entité, il s'agit de la programmation par sujets. Dans ce modèle, les préoccupations sont encapsulées par des sujets qui correspondent chacun à des classes. Ce modèle permet d'avoir une double structuration de l'application. Cette double structuration n'est pourtant utile que dans le cas de la réutilisation *a posteriori* comme par exemple dans [OHBS 94]. La double structuration permet dans ce cas de restructurer le programme avec des ensembles de classes différents (qui sont alors des sujets), ce qui permet de faire une première séparation sur les classes.

Selon nous, il n'est pas nécessaire d'avoir une double structuration de l'ensemble de classes, car nous pensons que restructurer un programme *a posteriori* pour y extraire des préoccupations et les réutiliser dans d'autres contextes est une activité plus complexe que de réécrire ces mêmes préoccupations.

Au cours du chapitre 3 nous avons pu encapsuler les préoccupations étudiées dans des ensembles de classes. Ces différents ensembles de classes étaient complètement découplés les uns des autres.

En conclusion, nous proposons que le modèle représente chaque préoccupation par un ensemble de classes. Nous étudions dans les parties suivantes comment composer à nouveaux ces ensembles de classes découplées.

4.2.2 Modélisation de la composition

Le paradigme de l'objet seul ne permet pas d'effectuer les opérations d'adaptation nécessaires pour composer les préoccupations (voir section 3.1.1). Pour pallier a ce problème la plupart des approches étudiées (voir section 2.3) proposent des extensions pour encapsuler la composition des préoccupations. La programmation par aspects introduit la notion d'aspect et la programmation par sujets les modules de composition pour encapsuler la composition des préoccupations.

Au cours de l'étude sur la réutilisation des préoccupations (voir chapitre 3) nous avons constaté que la composition représente la principale difficulté lorsque l'on considère la séparation des préoccupations.

La composition d'une préoccupation correspond à l'ensemble des adaptations qui permet de l'intégrer avec le reste de l'application. Néanmoins, même si chaque composition d'une même préoccupation change en fonction du contexte (c'est-à-dire de l'application), une partie de cette opération de composition reste la même.

Notre étude met en évidence cette propriété. Nous avons choisi de nommer *protocole de composition* la partie fixe de la composition d'une préoccupation. Nous avons montré dans le chapitre

3 que la présence de protocoles de composition permet de simplifier et de guider le travail d'adaptation nécessaire à la réutilisation des préoccupations.

Il est ainsi nécessaire de pouvoir encapsuler et abstraire une composition ; un protocole de composition est une abstraction de la composition d'une préoccupation. Comme l'encapsulation doit s'adapter aux différents contextes de réutilisation, l'entité encapsulant la composition doit aussi supporter une forme d'héritage.

Etant donné que notre objectif est de réaliser un modèle dans le cadre du paradigme objet, l'encapsulation de la composition doit être réifiée en une sorte de classe supportant l'abstraction et l'héritage et dont les membres sont des opérations d'adaptation et des variables. L'instanciation de cette réification réalisera la composition.

Nous avons choisi de nommer *adaptateur* l'entité qui encapsule la composition des préoccupations, cette entité contient donc un ensemble d'adaptations. Les sections suivantes définissent le pouvoir d'expression qu'elle doit proposer. Il faudra considérer en particulier : les modes de composition, les types d'adaptation et un mécanisme permettant l'abstraction et l'héritage.

4.3 EXPRESSIVITE NECESSAIRE A LA COMPOSITION

4.3.1 Plusieurs modes de composition

L'état de l'art de la séparation des préoccupations (voir section 2.3) permet de distinguer la composition *in-situ* de la composition *ex-situ*.

La composition *in-situ* est une composition asymétrique qui s'adresse à la réutilisation de préoccupations qui correspondent à des services offerts à des clients. Ce mode de composition modifie directement les clients pour y intégrer la préoccupation.

La composition *ex-situ* est une composition symétrique qui placera au même niveau les préoccupations à composer. Ce mode permet de garder inchangées les préoccupations à composer car leur composition génère une nouvelle préoccupation. Cette nouvelle préoccupation ainsi obtenue peut ensuite être utilisée comme toute autre préoccupation. La composition *ex-situ* s'appuie sur la génération de programme.

L'étude sur la réutilisation des préoccupations (voir le chapitre 3) montre que les deux modes de composition doivent être disponibles pour un support complet de la séparation des préoccupations dans le paradigme objet. La réutilisation des préoccupations non fonctionnelles (voir section 3.1) et des patrons de conception (voir section 3.3) nécessite un mode de composition *in-situ*. Tandis que la réutilisation des préoccupations fonctionnelles nécessite un mode de composition *ex-situ* (voir section 3.2).

Il est donc primordial que le modèle proposé supporte les deux modes de composition (voir section 3.4) d'autant plus qu'aucun modèle ou langage actuel ne possède ces deux modes de composition.

Le choix du mode de composition doit être effectué dans les adaptateurs. Comme ceux-ci sont des entités de première classe, ils peuvent appartenir à des ensembles de classes. C'est donc l'existence de l'ensemble de classes (noté paquetage) d'appartenance qui détermine le mode de composition : si l'ensemble de classes existe alors c'est *in-situ*, autrement c'est *ex-situ* et la composition génère alors le nouvel ensemble de classes.

4.3.2 Plusieurs types d'adaptation

De manière générale la composition d'une préoccupation nécessite des adaptations à la fois d'elle-même et du reste de l'application. Nous tenons à rappeler que dans notre terminologie un adaptateur encapsule un ensemble d'adaptations.

L'étude du chapitre 3 met en évidence l'ensemble complet des adaptations qui sont nécessaires à une bonne composition des préoccupations dans le paradigme de l'objet ; nous les rappelons ci-après :

1. modification d'une classe pour l'implémentation de nouvelles interfaces
2. fusion de classes et de méthodes
3. ajout ou redéfinition de méthodes dans des classes
4. ajout de nouvelles variables d'instance (et de classe) à des classes
5. interception avant, après, autour, sur exception des méthodes (qu'elles soient d'instance ou de classe)
6. interception des accès aux variables d'instances ou de classe

Les résultats de la section 3.4 montrent qu'aucun modèle ou langage actuel ne supporte ces six types d'adaptation à la fois. Notons que l'adaptation numérotée 1 pourra être généralisée à l'introduction de nouvelles super-classes dans un langage possédant l'héritage multiple. Chaque type d'adaptation et ses effets sont présentés par la suite.

4.3.2.1 Implémenter de nouvelles interfaces

L'adaptation n°1 permet d'opérer des changements dans une classe pour qu'elle implémente de nouvelles interfaces. Nous proposons de ne pas l'étendre à l'ajout de relations d'héritage car l'étude du chapitre 3 montre que l'héritage multiple n'apporte (dans le cas de la réutilisation de préoccupations) aucune solution viable par rapport à l'héritage simple. Cette adaptation utilise comme cible un point de jointure représentant des classes. Le mode de composition détermine l'effet produit : le mode *in-situ* modifie directement les classes cibles, le mode *ex-situ* produit de nouvelles classes qui incorpore l'adaptation.

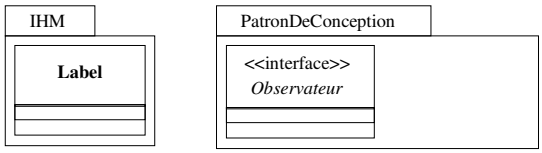


Figure 6. *Exemple utilisée pour l'adaptation n°1.*

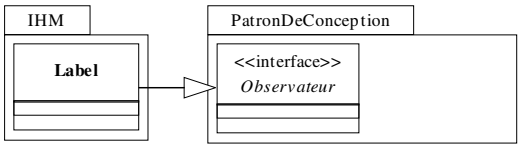


Figure 7. *Résultat de l'adaptation n°1 en mode in-situ*

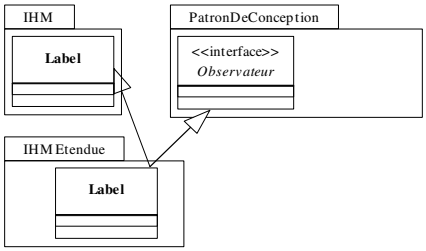


Figure 8. *Résultat de l'adaptation n°1 en mode ex-situ*

Les figures 6, 7, et 8 présentent un exemple de l'effet de l'adaptation n°1 en fonction du mode de composition. Les classes utilisées par l'adaptation sont décrites par la figure 6. L'adaptation est utilisée pour que la classe `Label` implémente désormais l'interface `Observateur`. La figure 7 présente le résultat de la composition en mode *in-situ* et la figure 8 le résultat en mode *ex-situ*.

4.3.2.2 Fusioner des classes

L'adaptation n°2 doit permettre de fusionner des classes de manière récursive. Cette opération est proche du résultat d'une relation d'héritage sauf qu'elle permet d'éviter de modifier directement les classes. Cette adaptation est le plus souvent utilisée en mode *ex-situ* (vois section 3.2.4). Elle doit permettre aussi de fusionner des implémentations de méthodes. Le mode de composition est soit une fusion asymétrique avec la composition *in-situ* et dans ce cas des classes doivent être spécifiées comme cible de la fusion, soit une fusion symétrique avec la composition *ex-situ* et dans ce cas un nouvel ensemble de classes est produit. Cette adaptation utilise deux points de jointure représentant deux ensembles distincts qui contiennent soit des classes, soit des méthodes. Un point de jointure représente les cibles de la fusion (classes ou méthodes où vont être fusionnées les sources). Le deuxième point de jointure représente donc les sources de la fusion.

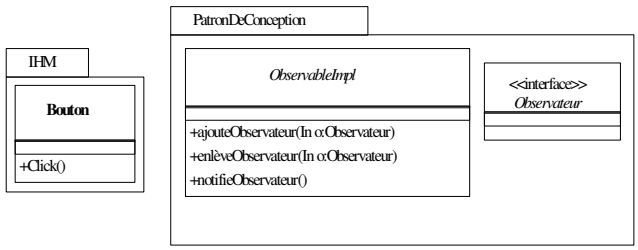


Figure 9. *Exemple utilisée pour l'adaptation n°2*

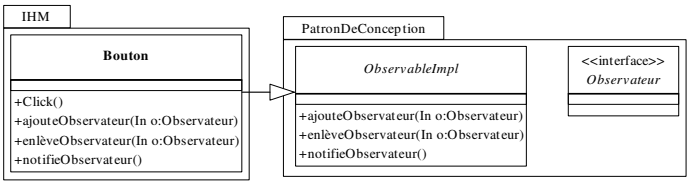


Figure 10. *Résultat de la composition de l'adaptation n°2 en mode in-situ*

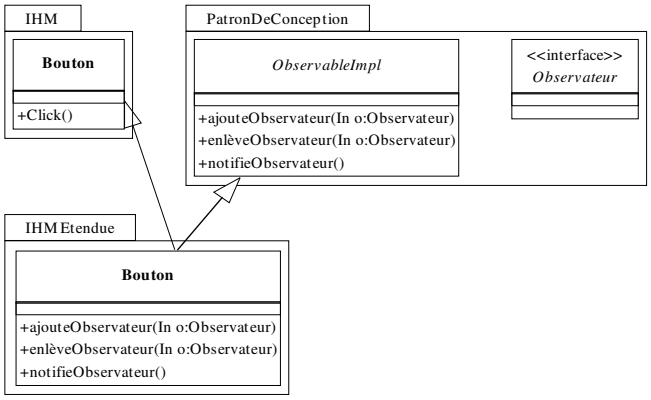


Figure 11. *Résultat de la composition de l'adaptation n°2 en mode ex-situ*

Les figures 9, 10, et 11 présentent un exemple de l'adaptation n°2. Les classes utilisées par l'adaptation sont décrites par la figure 9. L'adaptation n°2 est utilisée pour que la classe `Bouton` utilise l'implémentation fournie par la classe `ObservableImpl`. La figure 10 présente le résultat de la composition en mode *in-situ* et la figure 11 le résultat en mode *ex-situ*.

4.3.2.3 Ajout de membres

L'adaptation n°3 a vocation à représenter un mécanisme d'ouverture des classes qui permet d'ajouter ou de redéfinir des méthodes d'instance ou de classe. Ses cibles sont donc des classes qui sont contenues par un point de jointure. Tout comme les adaptations précédentes le mode de composition influence la sémantique de l'adaptation.

L'adaptation n°4 complète l'adaptation n°3 pour les variables d'instance et de classe.

4.3.2.4 Interceptions

L'adaptation n°5 est une adaptation fonctionnelle qui doit permettre d'ajouter du code avant, après, ou autour des méthodes. Les cibles de cette adaptation sont des méthodes qui n'appartiennent pas forcément aux mêmes classes. Tout comme les adaptations précédentes le mode de composition influence la sémantique de l'adaptation.

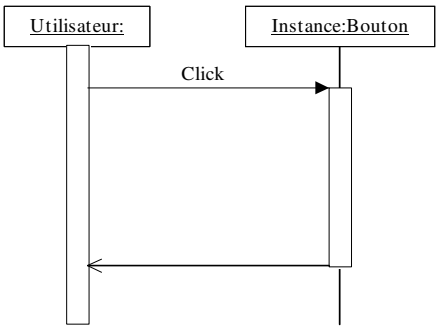


Figure 12. *Exemple utilisé pour l'adaptation n°5*

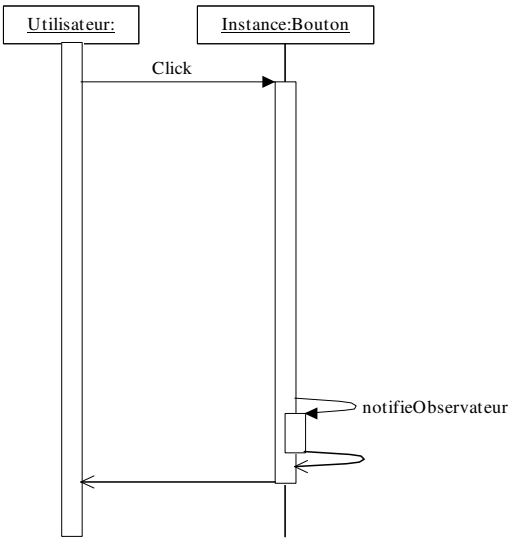


Figure 13. *Résultat de l'adaptation n°5 en mode in-situ*

Les figures 12 et 13 présentent un exemple pour l'adaptation n°5. Cette adaptation est utilisée dans l'exemple (voir figure 12) pour modifier la méthode Click de la classe `Bouton` pour que la méthode `notifyObservateur` soit appelée après chaque appel à `Click`. La figure 13 présente le résultat de l'adaptation en mode in-situ. La version ex-situ n'est pas présentée car le seul changement est que la classe `Bouton` modifiée est contenue par un autre paquetage, qui sera par exemple `IHMEtendue` (voir figure 11).

L'adaptation n°6 adresse les mêmes objectifs que l'adaptation n°5 mais concerne les accès aux variables d'instance ou de classe.

Nous allons maintenant voir plus en détail les capacités d'abstraction et de réutilisation des adaptateurs.

4.4 ABSTRACTION ET REUTILISATION DE LA COMPOSITION

Un adaptateur est composé d'un ensemble d'adaptations parmi celles que nous venons de décrire. Chaque opération d'adaptation utilise au moins un point de jointure pour représenter ses cibles (qui sont en fonction du type de l'adaptation soit des classes, soit des méthodes, ou soit des variables d'instance). Certaines adaptations, comme celle de type fusion, nécessitent un deuxième point de jointure qui représente les sources de la fusion.

L'étude du chapitre 3 suggère que pour faciliter la réutilisation des préoccupations, chaque préoccupation puisse être accompagnée de son protocole de composition. Cette propriété réduit l'étape de réutilisation d'une préoccupation à la spécialisation du protocole de composition pour le contexte de réutilisation.

Un protocole de composition est une abstraction des adaptations à faire pour réutiliser une préoccupation. Les protocoles de composition doivent être encapsulés par des adaptateurs abstraits (sorte de classes abstraites).

Les adaptations d'un adaptateur abstrait ou non nécessitent des points de jointure. Un point de jointure permet de décrire des cibles, or ces cibles peuvent être partagées entre différentes adaptations (d'un même adaptateur). Pour permettre aux adaptations de s'abstraire, leurs points de jointures doivent être plus flexibles. Nous avons choisi de permettre aux adaptateurs de posséder des variables (qui peuvent représenter elles aussi des cibles potentielles). Ces variables vont donc être utilisées lors de la description des points de jointure pour partager l'information qu'elles contiennent. En outre, ceci permet d'abstraire les points de jointure.

Un adaptateur abstrait doit donc être composé d'un ensemble d'adaptations (qui ne sont pas forcément toutes abstraite) et de variables (qui n'ont pas forcément toutes une valeur) ; en outre, il ne réalisera aucune composition.

Comme les protocoles de composition doivent être spécialisables (de manière à les utiliser par héritage pour réutiliser la préoccupation qu'ils représentent) et réalisables, l'adaptateur devra donc supporter la relation d'héritage simple. Comme les classes, les adaptateurs doivent pouvoir hériter d'un adaptateur abstrait ou concret, cette relation d'héritage devra supporter la redéfinition, la surcharge, et la concrétisation. En opposition à un adaptateur abstrait, un adaptateur concret réalisera effectivement la composition qu'il encapsule.

Les propriétés qui viennent d'être énumérées doivent être reflétées par les adaptations et les variables car ce sont les entités composantes des adaptateurs.

Une variable abstraite est représentée par un type qui est : soit un ensemble de classes, soit un ensemble de méthodes (d'instance ou de classe), ou soit un ensemble de variables (d'instance ou de classe). Si la variable est abstraite alors elle ne possède pas de valeur et ne représente aucune cible mais juste un type de cible. A l'inverse, si la variable est concrète alors il faut qu'elle spécifie ses cibles.

Chaque variable doit aussi posséder un nom unique à l'intérieur de l'adaptateur qui le contient ; ceci permet, par analogie avec les méthodes, de supporter la redéfinition de sa valeur dans les sous-adaptateurs.

Ce double système de variables et de points de jointure contenu par les adaptions offrira donc un support complet pour l'utilisation des protocoles de composition. En effet, il devient possible de spécifier une adaptation concrète qui possède un point de jointure faisant référence à une variable abstraite. Nous allons voir maintenant comment les adaptations doivent offrir aussi leur support aux protocoles de composition.

Une adaptation abstraite n'effectue aucune composition et nécessite que l'adaptateur qui la contient soit abstrait lui aussi. Rappelons qu'une adaptation quel que soit son type nécessite un point de jointure pour désigner ses cibles. Si on laisse libres (ou que l'on utilise une variable abstraite) les cibles d'une adaptation alors on permet de spécialiser cette dernière par héritage. Cette spécialisation est obligatoire à tout utilisateur du protocole de composition, dans cette optique l'utilisation des variables abstraites permet dans les cas idéaux de n'avoir qu'à fournir une valeur à ces variables.

Pour redéfinir ou surcharger une adaptation, il est nécessaire que cette dernière soit nommée et que son nom soit unique dans l'adaptateur. Ces deux propriétés permettent aux adaptations de supporter l'encapsulation des protocoles de composition et la concrétisation de ces derniers.

Les adaptateurs tels qu'ils viennent d'être décrits offrent un support à l'utilisation des protocoles de composition pour faciliter la réutilisation des préoccupations. Ceci est possible car les adaptateurs utilisent des propriétés du paradigme de l'objet l'héritage et la redéfinition.

4.5 CONCLUSION

Nous avons choisi après l'étude qualitative de la réutilisation des préoccupations dans le paradigme de l'objet (voir le chapitre 3) de construire un cahier des charges à partir des avantages et des manques constatés des différents modèles et langages étudiés (voir le Chapitre 2).

Pour parvenir à ce but, nous nous sommes inspirés de deux autres modèles : la programmation par aspects [KLM 97] et la programmation par sujets [HO 93]. Le Chapitre 3 montre que ces deux modèles – ensemble – possèdent presque toutes les fonctionnalités nécessaires à la séparation des préoccupations ; les fonctionnalités manquantes ont été énumérées dans le 3.4

Nous avons présenté dans ce chapitre le cahiers des charges pour faire de la réutilisation (en favorisant l'utilisation des protocoles de composition pour faciliter la réutilisation) de préoccupation dans le paradigme de l'objet. Pour y parvenir, chaque préoccupation doit être encapsulée par un ensemble de classes, ceci fournit la *séparation* des préoccupations. La composition des préoccupations doit être encapsulée par une nouvelle entité réifiée « l'adaptateur ». Cette nouvelle entité doit pouvoir s'abstraire, et posséder une relation d'héritage offrant ainsi la possibilité d'utiliser des protocoles de composition et de spécialiser ces deniers par héritage. L'adaptateur doit supporter toutes les adaptations, les points de jointure et les variables nécessaires au paradigme de l'objet (voir 3.5 pour un détail des nécessités de l'objet dans ce domaine). Nous allons donc présenter dans le chapitre suivant un nouveau modèle de réutilisation des préoccupations que nous avons construit à partir du cahier des charges présenté dans ce chapitre.

Chapitre 5

Le modèle

Le chapitre précédent présente le cahier des charges de notre modèle, ce dernier va être détaillé dans ce chapitre. Notre modèle de réutilisation des préoccupations est conçu pour les langages à objets à classes et il est indépendant de ces derniers même si l'on fait des hypothèses minimales. Les pré-requis et les langages les plus courants sont présentés dans la première section. La deuxième section présente une modélisation du langage source (noté MLS) utilisé pour décrire les préoccupations. La troisième section présente le concept introduit par notre modèle : l'adaptateur. La quatrième section présente une implémentation des opérateurs d'adaptation basée sur MLS.

5.1 PRE-REQUIS DES LANGAGES

Les langages utilisables par notre modèle doivent répondre à plusieurs pré-requis, ces derniers sont présentés dans le tableau ci-dessous :

	Java	**C++**	**Eiffel**	**C#**
Notion de classe	Oui			
Héritage entre classes	Simple	Multiple	Multiple	Simple
Encapsulation des méthodes d'instances dans les classes	Oui			
Variable d'instances	Oui			
Ensemble de classes	Package	namespace	clusters	namespace

Tableau 9. *Pré-requis sur les langages pour utiliser notre modèle.*

Notre modèle utilise les propriétés suivantes qui ne sont, elles, que facultatives : l'existence de variables de classe, l'existence de méthodes de classe, et des modificateurs de visibilité (utilisés pour les classes, les méthodes et les variables d'instance). Comme nous le remarquons les pré-requis sont minimaux et notre modèle peut donc être appliqué à la plupart des langages, le tableau 9 montre, par exemple, que quatre des principaux langages à objets à classes fournissent tous les éléments nécessaires.

5.2 MODELE DU LANGAGE SOURCE

Notre modèle de réutilisation des préoccupations peut être utilisé pour plusieurs langages sources comme le montre la section précédente. Le langage source représente le langage utilisé pour écrire le code source des applications et des préoccupations. Pour lui permettre d'être indépendant du langage source, notre modèle est défini au-dessus d'un modèle MLS générique utilisé pour unifier tous les langages source envisageables.

MLS est composé des entités suivantes : le paquetage, la classe, la méthode, l'attribut, la signature de méthode, la signature d'attribut, le corps de méthode, le modificateur de visibilité, et la liste (utilisée comme conteneur pour encapsuler des ensembles d'instances). Nous avons choisi d'utiliser la notion du tableau pour représenter les listes par soucis de visibilité. Ces différentes entités vont être présentées en utilisant le formalisme UML [UML 04].

Notons que MLS est extensible par héritage des entités qui le compose. Par exemple, il est envisageable de sous-classer `Method` pour prendre en compte le traitement des assertions du langage Eiffel.

Pour rester concis dans la description des différentes classes de MLS, nous n'allons pas parler des différentes méthodes jouant le rôle d'accesseurs et de modificateurs ; ces dernières étant nécessaires à la mise en œuvre du modèle.

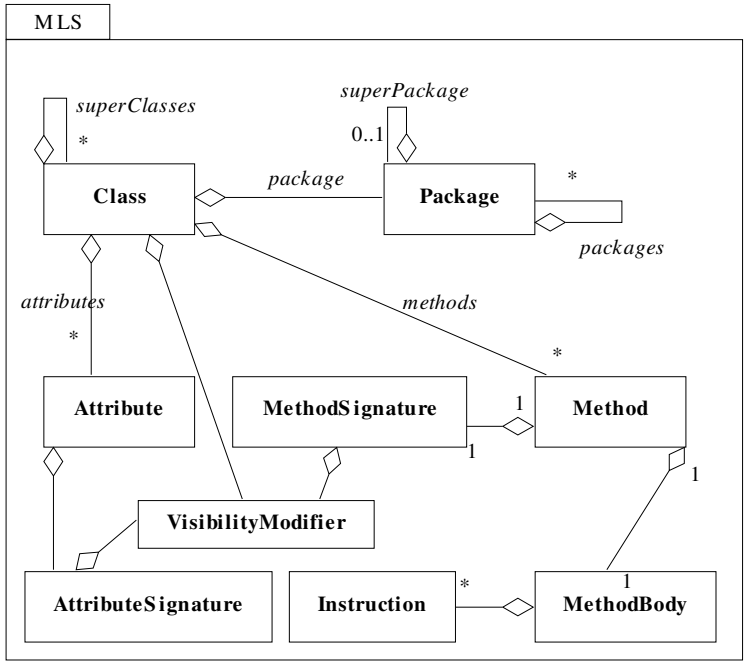


Figure 14. *Diagramme de classes de MLS*

La figure 14 présente toutes les classes qui composent MLS. MLS comporte deux principales entités d'encapsulation : la classe et le package. Le détail d'une classe est décomposé en attributs et méthodes. Chacun de ces deux derniers possède une signature (qui correspond à son nom, etc.). La méthode possède aussi un corps qui contient des instructions.

5.2.1 La classe

La classe représente une des deux principales entités d'encapsulation de MLS. Elle correspond à la notion de classe que l'on trouve dans les langages à objets.

<<Class>> Class
- name : String - package : Package - superClasses : Class[*] - methods : Method[*] - attributes : Attribute[*] - visibilityModifier : VisibilityModifier
+ setName(name : String) + getName() : String + setPackage(package : Package) + getPackage() : Package + setSuperClasses(superClasses : Class[*]) + getSuperClasses() : Class[*] + setMethods(methods : Method[*]) + getMethods() : Method[*] + setAttributes(attributes : Attribute[*]) + getAttributes() : Attribute[*] + setVisibilityModifier(vM : VisibilityModifier) + getVisibilityModifier() : VisibilityModifier + getMethod(method : MethodSignature) : Method + getAttribute(attribute : AttributeSignature) : Attribute + generateUniqueId(prefix : String) : String + clone () : Class

Diagramme 1. *La classe Class*

Une classe (voir diagramme 1) possède un nom unique dans le paquetage auquel elle appartient. Elle contient deux ensembles : un ensemble de méthodes et un ensemble d'attributs. Enfin, elle possède un modificateur de visibilité. Toutes ces variables d'instance nécessitent des accesseurs et des modificateurs car les opérateurs d'adaptation de notre modèle peuvent modifier chacune de ces variables. La méthode `getMethod()` retourne, si elle existe, la méthode qui possède la signature passée en argument. La méthode `getAttribute()` fait de même avec les attributs. La méthode `generateUniqueId()` génère un nouvel identificateur unique (utilisable pour un nom de méthode ou d'attribut) à partir d'un préfixe ; ceci sera utilisé pour masquer des méthodes ou des attributs en changeant leur nom.

5.2.2 Le paquetage

Le paquetage est la seconde entité d'encapsulation de MLS, il contient des classes. Un paquetage peut éventuellement contenir d'autres paquetages. Les paquetages sont donc organisés de manière hiérarchique depuis un paquetage spécial (`root`) qui est la racine de la hiérarchie.

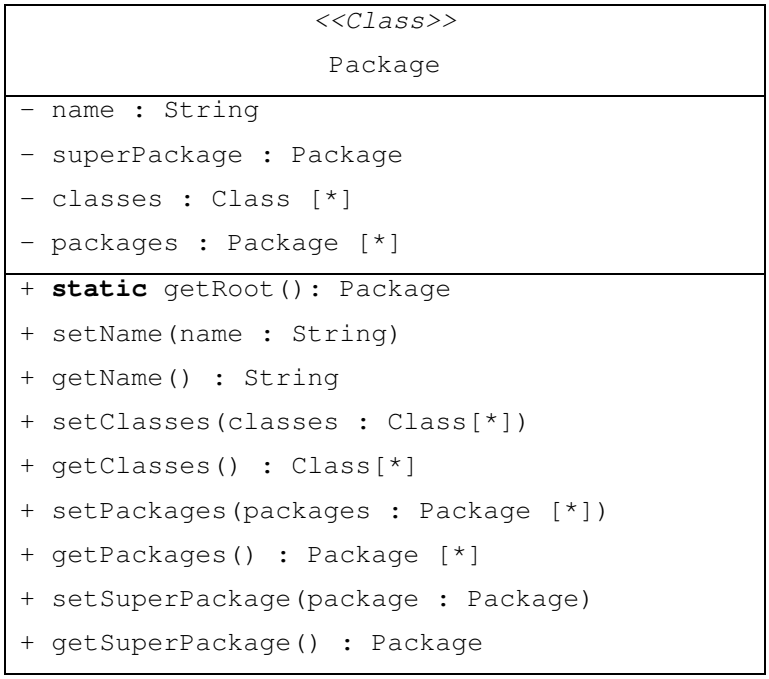


Diagramme 2. *La classe Package*

Un paquetage (voir diagramme 2) possède un nom (unique dans le paquetage auquel il appartient), un ensemble de `classes`, et un ensemble de `packages` (ses sous-paquetages). Il contient aussi une référence au paquetage qui le contient, et il possède un moyen d'obtenir le paquetage racine (`root`). Les autres méthodes sont des accesseurs et des modificateurs nécessaires à la mise en œuvre du modèle.

Le paquetage spécial `root` est un paquetage qui est son propre conteneur. Il représente le paquetage par défaut quand aucun paquetage n'est spécifié pour une classe.

5.2.3 L'attribut

Un attribut représente une variable associée à une classe.

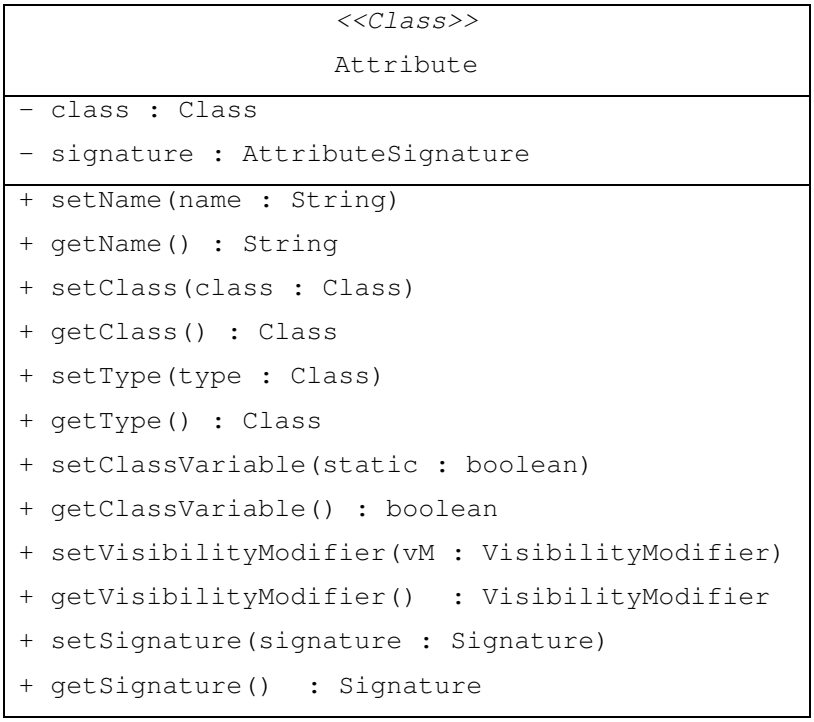


Diagramme 3. *La classe Attribute*

Un attribut (voir diagramme 3) possède une référence vers la classe qui le contient et une autre vers sa signature. La signature d'un attribut (voir section 5.2.4) correspond entre autres au nom de l'attribut, à une référence vers la classe qui représente son type, nous pouvons aussi distinguer si c'est un attribut de classe ou d'instance. La classe `Attribute` possède aussi des accesseurs et des modificateurs nécessaires à la mise en œuvre du modèle, la plupart correspondent à la signature de l'attribut.

5.2.4 La signature d'un attribut

Chaque attribut d'une classe possède une signature qui permet de l'identifier de manière unique.

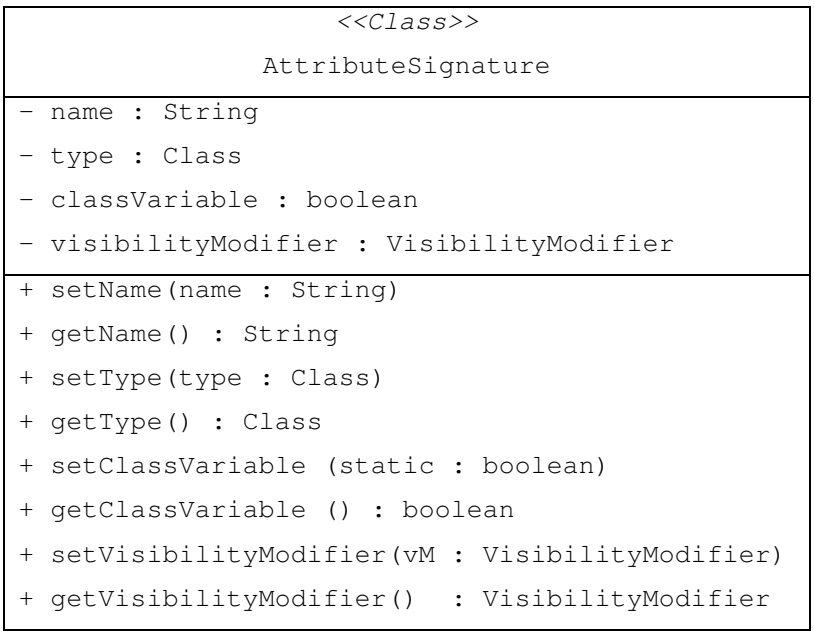


Diagramme 4. *La classe AttributeSignature*

La signature d'un attribut (voir diagramme 4) contient tous les éléments permettant de représenter l'attribut de manière unique à l'intérieur d'une classe. Il s'agit du nom de l'attribut, d'une référence vers la classe qui représente son type, d'un modificateur de visibilité, et d'une variable booléenne pour différencier les attributs de classe des attributs d'instance.

5.2.5 La méthode

La méthode est un message qui peut être envoyé vers une classe pour qu'elle effectue une action.

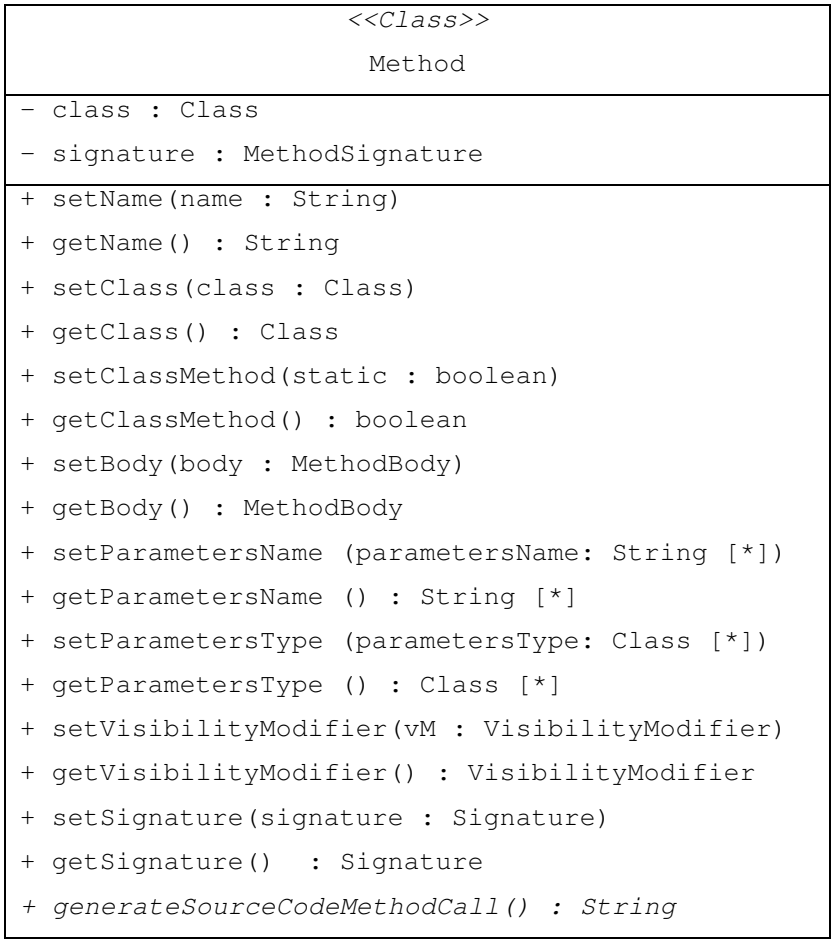


Diagramme 5. *La classe Method*

Une méthode (voir diagramme 5) possède un nom (unique dans l'espace de nom de la classe à laquelle elle appartient). Elle est définie par une référence vers la classe qui la contient, le corps de la méthode, et une signature. La signature d'une méthode (voir section 5.2.6) est caractérisée par un type de retour, un modificateur de visibilité, et les noms et types des paramètres de la méthode. La classe `Method` possède aussi des accesseurs et des modificateurs nécessaires à la mise en œuvre du modèle, la plupart concerne sa signature. La méthode abstraite `generateSourceCodeMethodCall()` a vocation à produire une chaîne de caractères utilisable pour faire un appel récursif à la méthode dans le langage source utilisé, elle est utilisée par les adaptations de type fusion (voir section 5.3.4.3).

5.2.6 La signature de méthode

La signature d'une méthode représente de manière unique une méthode.

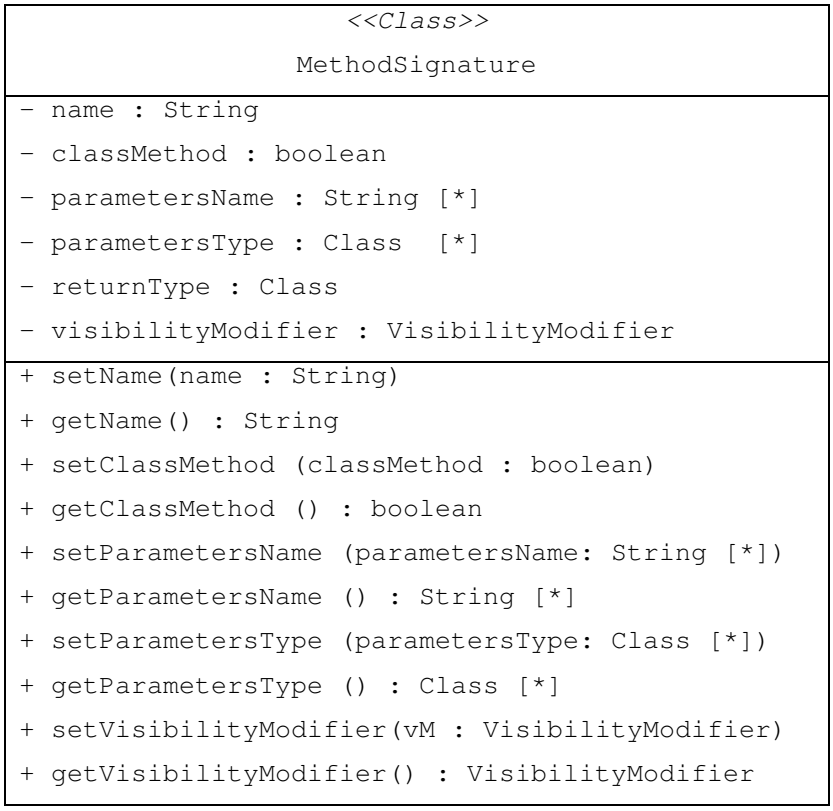


Diagramme 6. *La classe MethodSignature*

La signature d'une méthode (voir diagramme 6) se caractérise par un nom, un type de retour, un modificateur de visibilité, et les noms et types des paramètres (dont la quantité peut varier de zéro à plusieurs) de la méthode.

5.2.7 Le corps de méthode

Le corps d'une méthode est l'ensemble des instructions contenues dans la méthode.

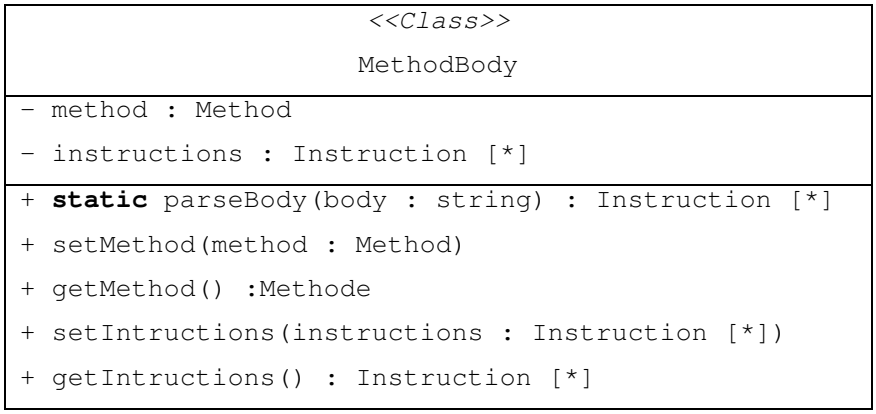


Diagramme 7. *La classe MethodBody*

Le corps d'une méthode (voir diagramme 7) est caractérisé par une référence vers la méthode associée et une liste d'instructions. La méthode de classe `parseBody()` permet de construire une liste d'instructions à partir du code source d'un corps de méthode.

Notons que notre modèle ne nécessite pas de description de la classe `Instruction` car les opérations d'adaptation opérées par notre modèle consistent seulement à ajouter (au début ou à la fin) des instructions à la liste d'instructions d'une méthode.

5.2.8 Le modificateur de visibilité

Un modificateur de visibilité peut s'appliquer aux classes, aux méthodes, et aux attributs.

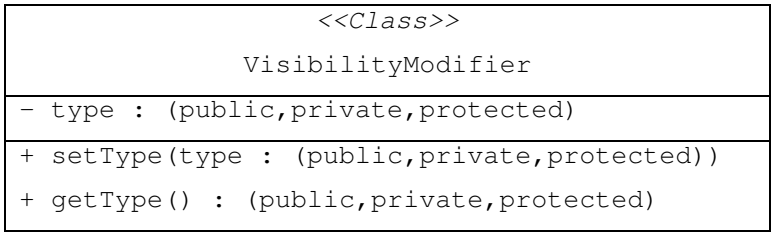


Diagramme 8. *La classe VisibilityModifier*

Un modificateur de visibilité (voir diagramme 8) peut prendre l'une des trois valeurs suivantes : `public` (visible par tout le monde), `private` (visible uniquement par le conteneur) ou `protected` (visible uniquement par le paquetage et ses sous-paquetages).

5.3 NOUVELLES ENTITES INTRODUITES PAR NOTRE MODELE

L'adaptateur est une des principales entités introduites par notre modèle. Cette entité est de première classe et permet d'encapsuler des adaptations. Un adaptateur contient un ensemble de variables et d'adaptations. La section 5.3.1 présente l'adaptateur, la section 5.3.2 présente les variables d'un adaptateur, la section 5.3.4 présente les adaptations possibles, la section 5.3.3 présente les points de jointure.

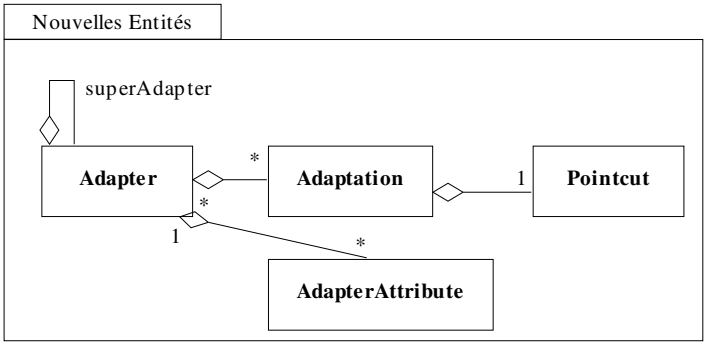


Figure 15. *Nouvelles entités introduites par le modèle*

La figure 15 présentes les nouvelles entités introduites par notre modèle. L'*adapter* est la nouvelle entité d'encapsulation. Elle contient des adaptations (pour le détails de la hiérarchie des adaptations voir section 5.4) et des variables.

5.3.1 L'adaptateur

L'adaptateur est l'entité de notre modèle qui encapsule la composition.

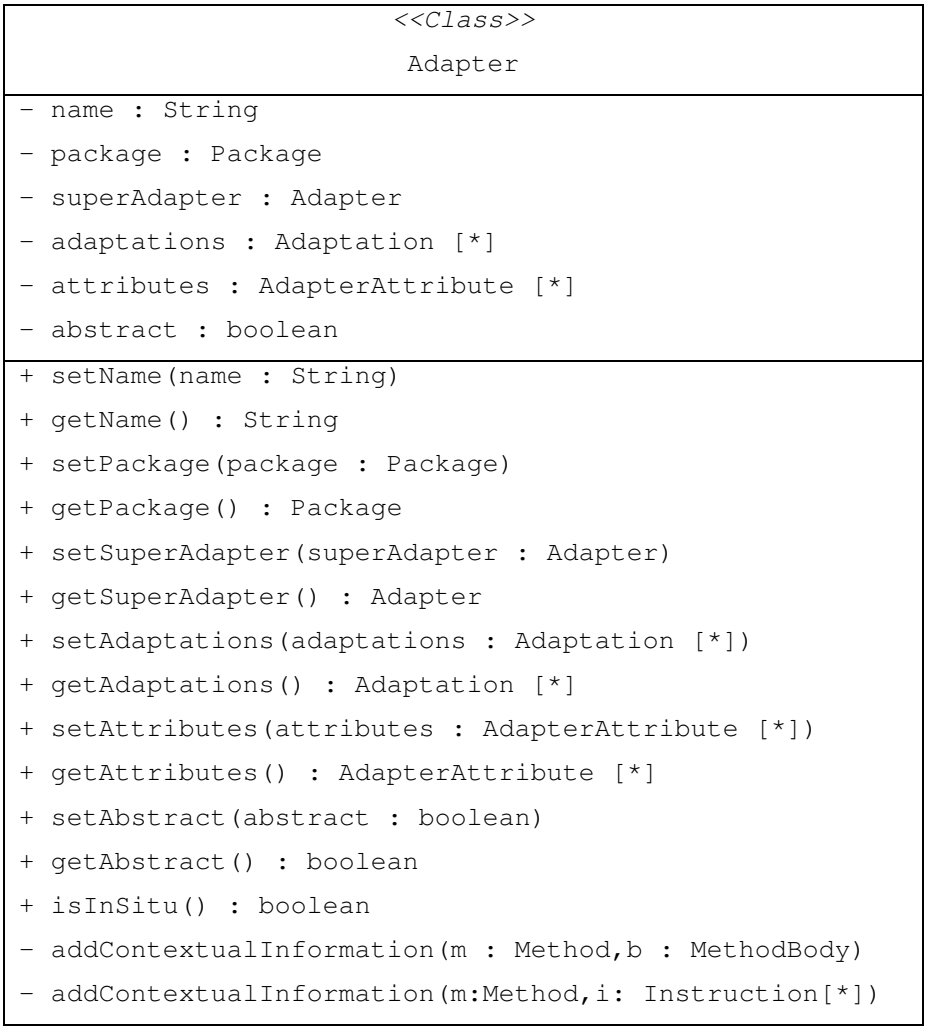


Diagramme 9. *La classe Adapter*

Un adaptateur (voir diagramme 9) possède un nom unique dans le paquetage auquel il appartient. Il contient aussi un ensemble d'adaptations et un ensemble d'attributs. De plus, il peut être abstrait. Enfin, il possède une référence vers son éventuel super-adaptateur. Les méthodes `addContextualInformation()` modifient le corps de la méthode pour implémenter le passage de contexte dans les adaptations de types interception (voir section 5.4.1). Cette modification peut se faire : soit par l'intermédiaire du `MethodBody`, soit de la liste des `Instruction`. Elle consiste à ajouter au début quatre variables contenant les informations contextuelles : le nom de la classe interceptée, le nom de la méthode interceptée et deux tableaux contenant les valeurs et les noms des arguments de la méthode interceptée.

La méthode `isInSitu()` (voir section 4.2.2 pour une définition de *in-situ* et *ex-situ*) permet de savoir si l'adaptateur opère en mode *in-situ*. Dans ce cas là, il opére directement les adaptations

qu'il contient sur ses cibles. Dans le cas contraire, l'adaptateur est dit ex-situ et les adaptations qu'il contient créent de nouvelles classes dans le paquetage `package`. Un adaptateur est in-situ si le paquetage qui le contient effectivement est le même que le paquetage qu'il déclare (représenté par la variable `package`), dans le cas contraire l'adaptateur est ex-situ.

5.3.2 Variable d'un adaptateur

Un adaptateur peut contenir des variables pour représenter des ensembles de classes ou de méthodes. Les variables sont utilisées lors de la description des points de jointure, ceci permet d'abstraire les points de jointure (voir section 4.2.2), ce qui permet d'abstraire un adaptateur et de lui faire jouer le rôle de protocole de composition.

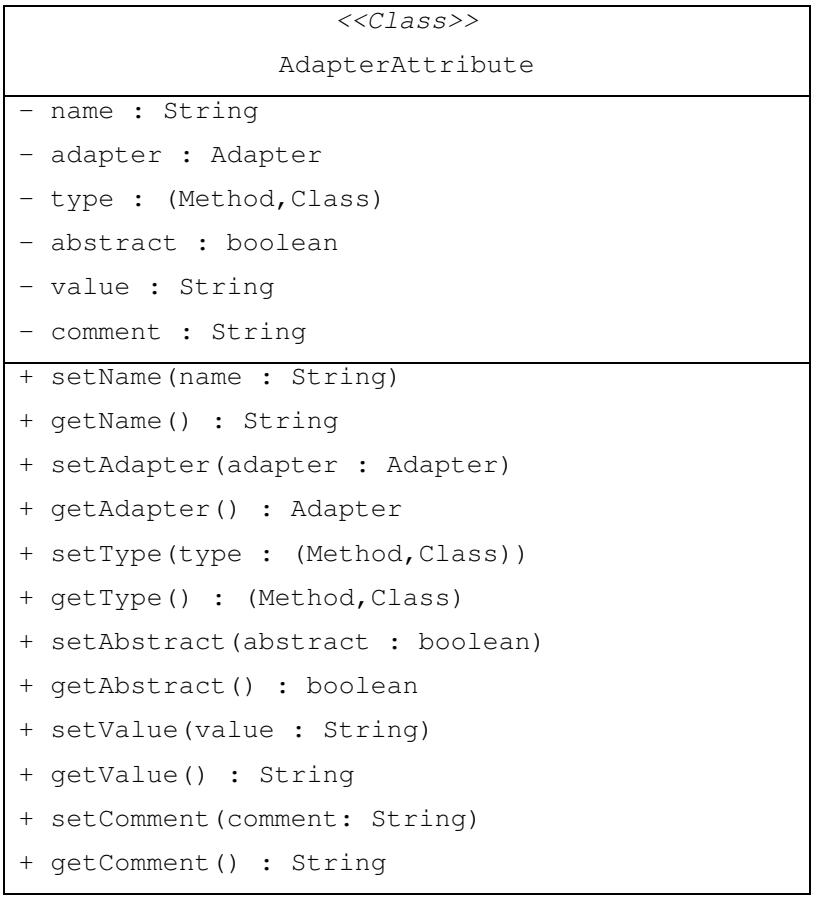


Diagramme 10. *La classe AdapterAttribute*

Une variable d'un adaptateur (voir diagramme 10) est caractérisée par un nom unique dans l'adaptateur auquel elle appartient, une référence vers l'adaptateur qui la contient, son type (méthode ou classe), un booléen qui signifie que l'attribut n'a pas de valeur (il représente alors une variable sans valeur), une valeur et éventuellement un commentaire.

5.3.3 Le point de jointure

Un point de jointure permet de décrire des ensembles de classes ou de méthodes à partir dune expression régulière sur les paquetages, les noms des classes et les signatures de méthodes.

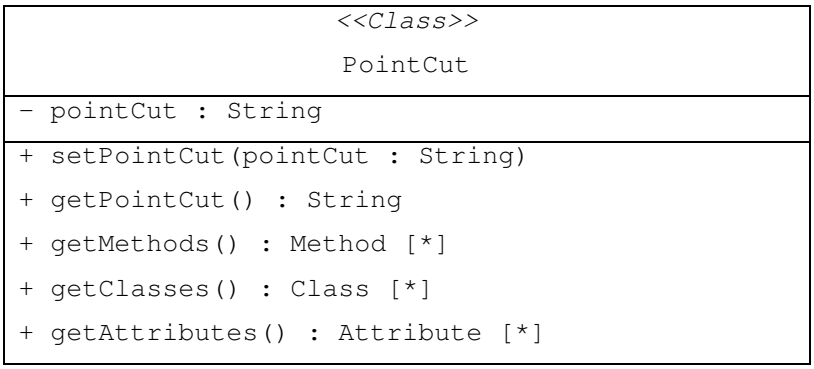


Diagramme 11. *La classe PointCut*

La classe `PointCut` (voir diagramme 11) contient une variable de type chaîne de caractères qui contient l'expression régulière qui décrit le point de jointure, cette expression régulière peut faire référence à des noms de variables provenant de l'adaptateur qui déclare le point de jointure. Des exemples de l'état de l'art d'expressions régulières pour les points de jointures sont présents dans le chapitre 3. Les méthodes `getMethodes()`, `getClasses()`, et `getAttributes()` permettent d'obtenir toutes les cibles du point de jointure en fonction des entités concernées. Les cibles du point de jointure sont décrites par le modèle MLS ; en effet les classes `Method`, `Class`, et `Attribute` proviennent de ce modèle. Le type (ensemble de classes ou de méthodes) d'un point de jointure est directement déterminé par les cibles représentées par l'expression régulière contenue par la variable `pointCut`.

5.3.4 Les adaptations

La classe `Adaptation` est la super-classe de toutes les adaptations supportées par le modèle.

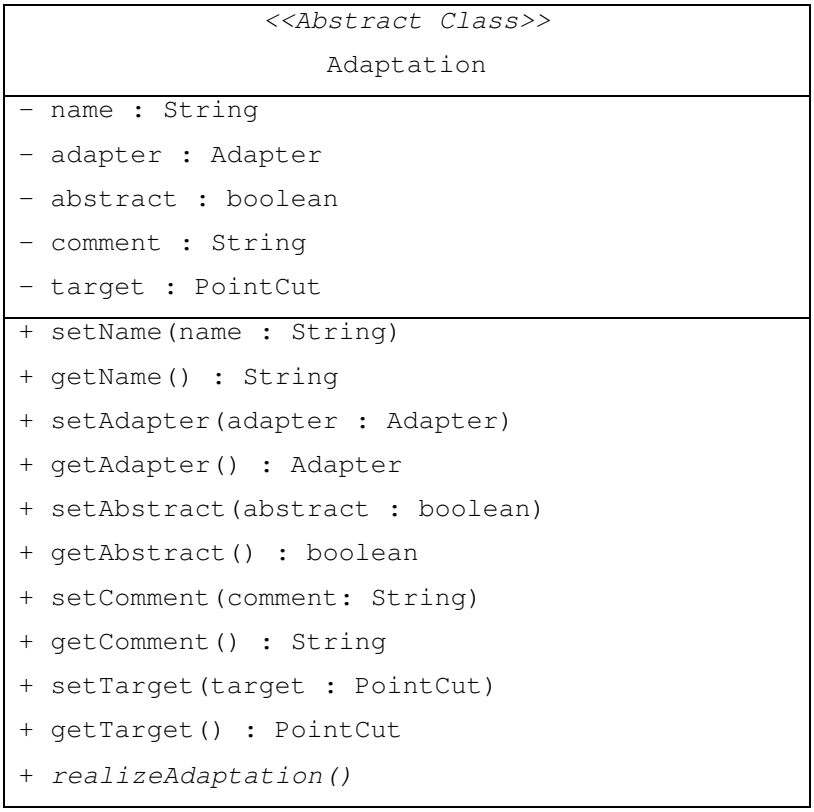


Diagramme 12. *La classe Adaptation*

Une adaptation (voir diagramme 12) possède un nom qui est unique pour l'adaptateur dans lequel elle est déclarée. Elle se caractérise par une référence vers l'adaptateur qui la contient, un booléen `abstract` signifiant que l'adaptation est abstraite, un point de jointure et éventuellement un commentaire. La méthode abstraite `realizeAdaptation()` effectue l'adaptation ; elle est concrétisée par les sous-classes.

Le point de jointure (voir section 5.3.3 pour plus de détails) `target` représente les cibles potentielles de l'adaptation. Ces cibles peuvent être des classes, des méthodes ou des variables. Les variables définies par l'adaptateur peuvent être utilisées lors de la description du point de jointure, ceci permet d'abstraire les points de jointure.

La hiérarchie des adaptations est présentée dans la figure 16. Ces adaptations sont présentées dans la suite de ce chapitre.

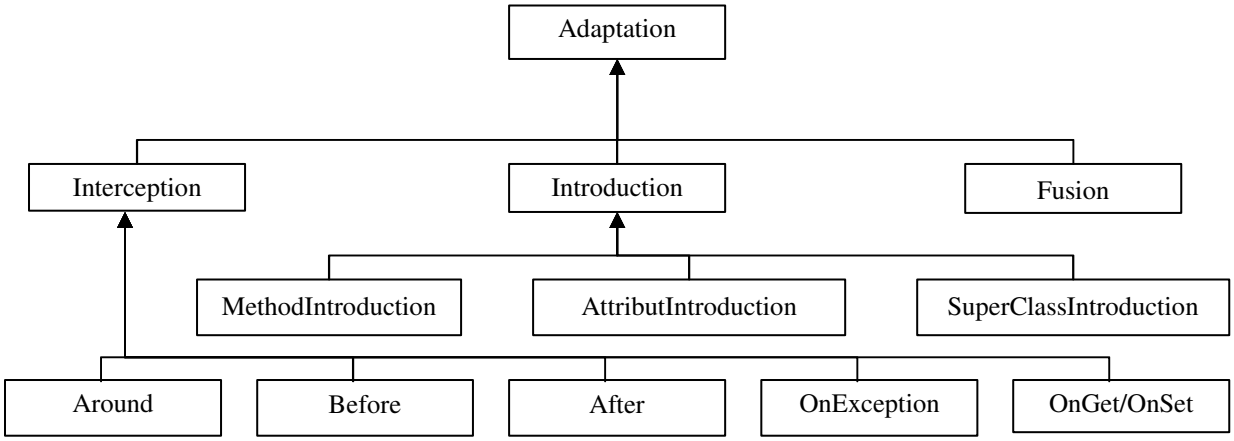


Figure 16. *Hiérarchie des Adaptations*

Le cahier des charges (voir section 4.3.2) spécifie les adaptations qui sont nécessaires pour la prise en charge complète de la composition des préoccupations dans les langages à objets à classes ; nous les rappelons ci-après :

1. Modification d'une classe pour l'implémentation de nouvelles interfaces
2. Fusion de classes et de méthodes
3. Ajout ou redéfinition de méthodes dans des classes
4. Ajout de nouvelles variables d'instance (et de classe) à des classes
5. Interception avant, après, autour, sur exception des méthodes (quelles soient d'instance ou de classe)
6. Interception des accès aux variables d'instance ou de classe

Les adaptations du type 5 et 6 sont mises en oeuvre par les sous-classes de la classe `Interception`, elles sont présentées dans la section 5.3.4.1 . Les adaptations de type 1[1], 3, et 4 sont réalisées par les sous-classes de la classe `Introduction`, elles sont présentées dans la section 5.3.4.2 . Enfin, les adaptations de type 2 sont gérées par la classe `Fusion` et présentées dans la section 5.3.4.3 .

Notons que toutes les adaptations modifient directement les classes cibles si l'adaptateur qui les contient est *in-situ* (voir `Adaptater.isInSitu()` section 5.3.1), autrement ces adaptations sont réalisées sur une copie (dans un autre paquetage, qui est alors représenté par `Adaptater.getPackage()`) des classes cibles.

[1] Généralisé à la modification des super-classes.

5.3.4.1 Adaptation de type Interception

Il y a deux catégories d'adaptation de type *interception*. La première concerne les méthodes et permet d'insérer un nouveau comportement qui pourra être exécuté : autour, avant, après, ou sur levée d'exception. La seconde s'adresse aux variables et permet d'exécuter du code lors de l'accès en lecture ou en écriture sur des variables d'instance ou de classe. L'ensemble des méthodes (ou variables) cibles est représenté par le point de jointure `target` de la classe `Adaptation`.

La classe `Interception` représente la super-classe de toutes les interceptions supportées par le modèle.

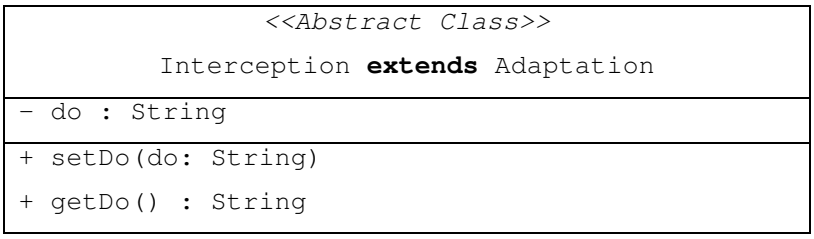


Diagramme 13. *La classe Interception*

La classe `Interception` (voir diagramme 13) contient une chaîne de caractère (`Do`) qui représente le code source à exécuter au point de jointure. `Do` représente donc l'adaptation comportementale qui doit être effectuée. Cette chaîne de caractère doit correspondre à un corps de méthode car elle est analysée par la méthode `MethodBody.parseBody()`.

Les sous-classes de la classe `Interception` ne sont pas détaillées car elles ne servent qu'à redéfinir la méthode `realizeAdaptation()` de la classe `Adaptation` en fonction du type de l'interception à réaliser. Ces adaptations ont été employées par exemple pour réutiliser une préoccupation : la « trace » (voir section 3.2). L'implémentation des différentes méthodes est présentée dans la section 5.4.1.

5.3.4.2 Adaptation de type Introduction

L'adaptation de type *introduction* permet d'ajouter : soit un attribut à un ensemble de classes, soit une méthode (ou de la redéfinir si elle existe déjà) à un ensemble de classes, soit enfin une nouvelle super-classe à un ensemble de classes. L'ensemble des classes cibles est représenté par le point de jointure `target` de la classe `Adaptation`.

La classe abstraite `Introduction` n'est pas présentée car elle est vide, elle n'a d'utilité que pour identifier les différentes adaptations du type *introduction*.

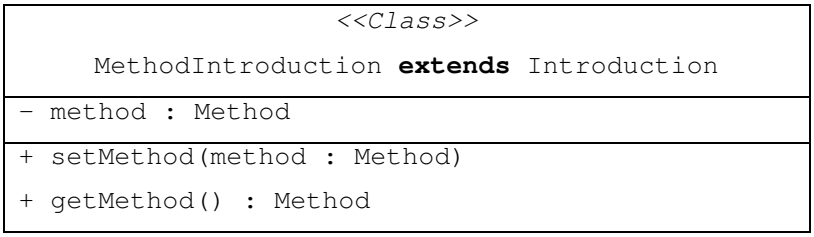


Diagramme 14. *La classe MethodIntroduction*

La classe `MethodIntroduction` (voir diagramme 14) contient la méthode à introduire dans les classes cibles. Notons, que si la méthode existe déjà il est aussi possible d'utiliser l'interception de type `Around` pour parvenir à un résultat similaire.

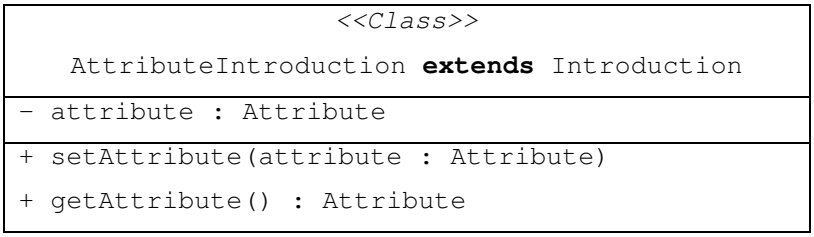


Diagramme 15. *La classe AttributIntroduction*

La classe `AttributeIntroduction` (voir diagramme 15) permet l'ajout d'attribut. Elle contient l'attribut à introduire dans les classes cibles.

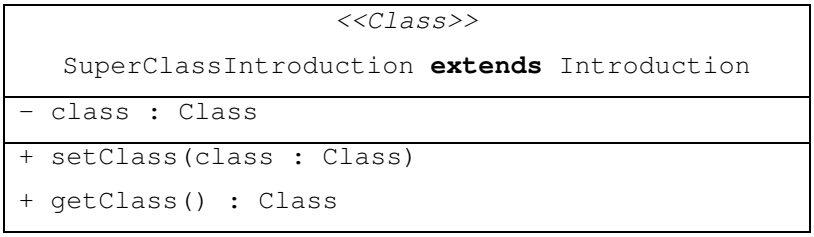


Diagramme 16. *La classe SuperClassIntroduction*

La classe `SuperClassIntroduction` (voir diagramme 16) correspond à l'ajout d'une nouvelle super-classe. Elle contient la classe à introduire à la liste des super-classes des classes cibles. Elle n'est utilisable que dans les langages à héritage multiple. Pour un langage à héritage simple, elle ne peut être utilisée que si la classe à adapter n'a pas de super-classe ; autrement il faut utiliser l'adaptation de type fusion.

L'implémentation de la méthode `realizeAdaptation()` de ces classes est présentée à la section 5.4.2.

5.3.4.3 Adaptation de type Fusion

L'adaptation de type fusion permet de fusionner soit un ensemble de classes dans une classe, soit un ensemble de méthodes dans une méthode. Cette adaptation est un palliatif à l'héritage multiple qui est particulièrement intéressant dans le cadre d'une composition *ex-situ*.

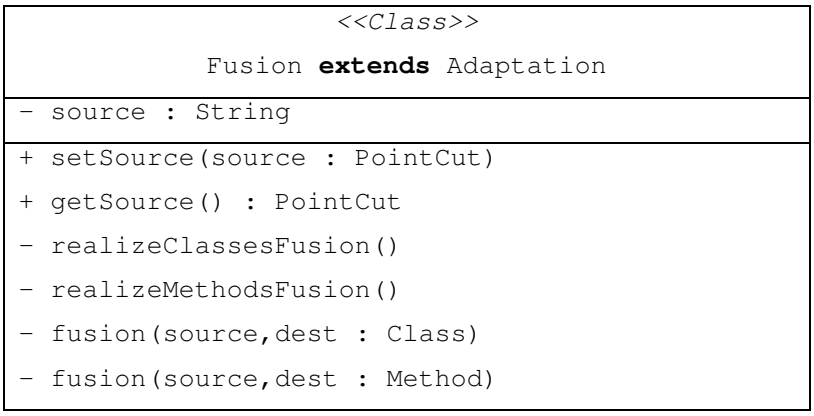


Diagramme 17. *La classe Fusion*

La classe `Fusion` (voir diagramme 17) contient un point de jointure qui représente soit des classes soit des méthodes. L'ensemble des classes (respectivement méthodes) défini par le point de jointure `source` est fusionné dans l'ensemble des classes (respectivement méthodes) définies par le point de jointure `target` (provenant de la classe `Adaptation`). Les méthodes `realize-ClassesFusion()`, `realize-MethodsFusion()` et `Fusion()` sont utilisées en interne et présentées dans la section 5.4.3. Cette adaptation a été utilisée par exemple pour le patron de conception (voir section 3.3).

5.4 IMPLEMENTATION DES ADAPTATIONS DANS LE MODELE MLS

Cette section présente, à l'aide d'un pseudo-langage de programmation qui manipulera les entités du modèle MLS, l'implémentation des adaptations mises en oeuvre par notre modèle. Les implémentations présentées ici correspondent à l'implémentation de la méthode `realize-Adaptation()` dans chacune des sous-classes de la classe `Adaptation`. Pour que le code reste lisible nous ne présentons que l'implémentation en mode *in-situ* des différentes adaptations, le mode *ex-situ* nécessite en supplément de dupliquer la classe qui va être modifiée et de la mettre dans le paquetage généré par l'adaptateur.

5.4.1 Implémentation des interceptions

Les interceptions mettent en œuvre des adaptations comportementales car elles permettent d'intercepter des appels de méthodes ou des accès à des attributs pour modifier le comportement existant.

5.4.1.1 Around

L'interception de type `Around` permet d'exécuter le code représenté par la variable `do` de l'interception autour du code original des méthodes ciblées par le point de jointure `target`.

```
1   Procedure Around.realizeAdaptation() {
2     foreach m : Method in target.getMethods() {
3       Class c = m.getClass();
4       //Hide original method
5       Method oldMethod = m.clone();
6       oldMethod.setName( c.generateUniqueId( oldMethod.getName() ) );
7       oldMethod.setVisibilityModifier( new VisibilityModifier ( private ) );
8       c.getMethods().add(oldMethod);
9       //Replace original method's body
10      Instruction[] newBody = MethodBody.parseBody(do);
11      addContextualInformation(m,newBody);
12      //Replace proceed() call by a call to oldMethod
13      m.getBody().setInstruction( newBody ) ;
14    }
15  }
```

Le code ci-dessus masque chacune des méthodes interceptées. Ansi la méthode interceptée est dupliquée et son duplicata change de nom (voir ligne 6) et devient privé (voir ligne 7). Le corps de la méthode originale est ensuite modifié pour correspondre au corps de méthode représentée par la variable `do` de l'adaptation (voir ligne 10 à 13). L'introduction de l'information contextuelle correspondante (voir section 5.3.1) est realisée à la ligne 11. Il faut noter que le corps de méthode `do` de `Around` utilise un mot-clef spécial `proceed()` pour représenter l'appel à la méthode originelle avant l'interception. La ligne 12 permet de remplacer l'appel à la pseudo-méthode `proceed()` par un appel à la méthode masquée `oldMethod`. Cette dernière modification s'occupe aussi de traiter le type éventuel de retour de la méthode.

5.4.1.2 Before

L'interception de type `Before` permet d'exécuter le code représenté par la variable `do` de l'interception avant le code original des méthodes ciblées par le point de jointure `target`.

```
1   Procedure Before.realizeAdaptation() {
2     foreach m : Method in target.getMethods() {
3       Instruction[] newBody = MethodBody.parseBody(do);
4       addContextualInformation(m,newBody);
5       m.getBody().getInstruction().insert(newBody,0) ;
6     }
7   }
```

Le code ci-dessus modifie la liste d'instructions de chaque méthode interceptée pour ajouter au début de cette liste les instructions contenues dans la variable `do` de l'adaptation. L'introduction de l'information contextuelle est réalisée par la ligne 4.

5.4.1.3 After

L'interception de type `After` permet d'exécuter le code représenté par la variable `do` de l'interception après le code original des méthodes ciblées par le point de jointure `target`. Bien évidemment l'implémentation de l'interception de type `After` est différente de l'interception `Before`, car l'interception de type `After` a lieu après l'appel de la méthode d'origine et doit prendre en compte une possible valeur de retour.

```
1   Procedure After.realizeAdaptation() {
2     foreach m : Method in target.getMethods() {
3       Class c = m.getClass();
4       //Hide original method
5       Method oldMethod = m.clone();
6       oldMethod.setName( c.generateUniqueId( oldMethod.getName() ) );
7       oldMethod.setVisibilityModifier( new VisibilityModifier ( private ) );
8       c.getMethods().add(oldMethod);
9       //Replace original method's body
10      Instruction[] newBody = MethodBody.parseBody(do);
11      addContextualInformation(m,newBody);
12      //Insert a call to oldMethod in the begenning of newBody
13      m.getBody().setInstruction( newBody ) ;
14    }
15  }
```

Le code ci-dessus masque chacune des méthodes interceptées. Pour cela la méthode interceptée est dupliquée, son duplicata change de nom (voir ligne 6) et devient privé (voir ligne 7). Le corps de la méthode originale est ensuite modifiée pour correspondre au corps de méthode représenté par la variable `do` de l'adaptation (voir lignes 10 à 13). L'introduction de l'information contextuelle est effectuée par la ligne 11. Enfin, le nouveau corps de la méthode est une dernière fois modifiée pour faire un appel à la méthode masquée au début (voir ligne 12). Le masquage de la méthode originelle simplifie l'implémentation car l'interception After est exécutée après l'appel de cette dernière (et doit donc retourner elle aussi une eventuelle valeur de retour).

5.4.1.4 OnException

L'interception de type `OnException` permet d'exécuter le code représenté par la variable `do` de l'interception autour du code originel des méthodes ciblées par le point de jointure `target` lorsque code originel a levé une exception.

```
1   Procedure OnException.realizeAdaptation() {
2     foreach m : Method in target.getMethods() {
3       Class c = m.getClass();
4       //Hide original method
5       Method oldMethod = m.clone();
6       oldMethod.setName( c.generateUniqueId( oldMethod.getName() ) );
7       oldMethod.setVisibilityModifier( new VisibilityModifier ( private ) );
8       c.getMethods().add(oldMethod);
9       //Replace original method's body
10      Instruction[] newBody = MethodBody.parseBody(do);
11      addContextualInformation(m,newBody);
12      //Insert a call to oldMethod surrounded by a try in the beginning of newBody
13      m.getBody().setInstruction( newBody ) ;
14    }
15  }
```

Le code ci-dessus masque chacune des méthodes interceptées. Pour cela la méthode interceptée est dupliquée, son duplicata change de nom (voir ligne 6) et il devient privé (voir ligne 7). Le corps de la méthode originale est ensuite modifié pour correspondre au corps de méthode qui est représenté par la variable `do` de l'adaptation (voir lignes 10 à 13). L'introduction de l'information contextuelle est effectuée par la ligne 11. Enfin, le nouveau corps de la méthode est une dernière fois modifié pour faire un appel à la méthode masquée au début (voir ligne 12), cet appel étant entouré d'un bloc `try` pour capturer les exceptions. Le code représenté par le `do` correspond donc à la partie `catch` du bloc `try/catch` construit par l'adaptation.

5.4.1.5 OnGet / OnSet

L'interception de type `OnGet/OnSet` permet d'exécuter le code représenté par la variable `do` de l'interception avant l'accès en lecture ou la modification d'un attribut d'une classe. Nous n'allons pas présenter l'implémentation ici car cette adaptation réalise des changements sur tous les endroits où est accédé l'attribut intercepté. De plus, le génie logiciel conseille d'utiliser des

accesseurs et de modificateurs pour éviter l'accès direct aux attributs des classes. Or, ces accesseurs et modificateurs sont des méthodes classiques qui sont interceptables par les autres adaptations présentées dans les section précédentes (After,Before, Around …).

Pour réaliser une interception `onGet` ou `onSet`, il faut introduire dans les classes à modifier soit un accesseur soit un modificateur. Ensuite, il faut transformer tous les accès en lecture ou en écriture à cette variable pour utiliser à la place soit l'accesseur, soit le modificateur.

5.4.2 Introduction

Les adaptations de type introduction mettent en oeuvre des adaptations fonctionnelles, elles permettent d'introduire dans des ensembles de classes soit de nouvelles méthodes, soit de nouveaux attributs soit une nouvelle super-classe.

5.4.2.1 Ajout et redéfinition d'une méthode

L'adaptation de type `MethodIntroduction` permet d'ajouter (ou de redéfinir si elle existe déjà) une nouvelle méthode (représentée par la variable `method`) dans les classes ciblées par le point de jointure `target`.

```
1   Procedure MethodIntroduction.realizeAdaptation() {
2     foreach c : Class in target.getClasses() {
3       Method oldMethod = c.getMethod(method.getSignature());
4       if ( oldMethod != null ) {
5         MethodBody newBody = method.getBody();
6         addContextualInformation(method,newBody);
7         oldMethod.setBody( newBody );
8       } else {
9         Method newMethod   = method.clone();
10        MethodBody newBody = newMethod.getBody();
11        addContextualInformation(method,newBody);
12        newMethod.setClass( c );
13        c.getMethods().add( newMethod );
14      }
15    }
16  }
```

Le code décrit ci-dessus modifie chacune des classes représentées par le point de jointure `target` (voir ligne 2). Cette modification permet d'introduire la méthode `method` appartenant à l'adaptation. Si la classe modifiée possède déjà une méthode ayant la même signature (voir lignes 3 à 4) alors le corps de cette méthode est remplacé par une copie du corps de la méthode `method` (voir ligne 5 à 7) ; ce dernier étant modifié pour contenir l'information contextuelle (voir ligne 6 et section 5.3.1). Si la méthode n'existe pas dans la classe modifiée, alors `method` est dupliquée (voir ligne 9) et modifiée pour ajouter l'information contextuelle dans son corps (voir ligne 11) et elle est ajoutée à la classe (voir ligne 12 à 13).

5.4.2.2 Ajout d'un attribut

L'adaptation de type `AttributeIntroduction` permet d'ajouter un attribut (représenté par la variable `attribute`) dans les classes ciblées par le point de jointure `target`.

```
1   Procedure AttributeIntroduction.realizeAdaptation() {
2     foreach c : Class in target.getClasses() {
3       Attribute oldAttribute = c.getAttribute(attribute);
4       if ( oldAttribute == null ) {
5         Attribute newAttribute= attribute.clone();
6         newAttribute.setClass( c );
7         c.getAttributes().add( newAttribute );
8       }
9     }
10  }
```

Le code modifie chacune des classes représentées par le point de jointure `target` (voir ligne 2). Cette modification permet d'ajouter l'attribut `attribute` défini par l'adaptation. Si la classe modifiée ne possède pas l'attribut, alors une copie de ce dernier (voir ligne 5) est ajoutée à la classe (voir lignes 6 à 7).

5.4.2.3 Ajout d'une super-classe

L'adaptation de type `SuperClassIntroduction` permet d'ajouter une nouvelle super-classe (représentée par la variable `class`) dans les classes ciblées par le point de jointure `target`.

```
1   Procedure SuperClassIntroduction.realizeAdaptation() {
2     foreach c : Class in target.getClasses() {
3       if (  ! c.getSuperClasses().contains(class) {
4         c.getSuperClasses().add( class );
5       }
6     }
7   }
```

Le code modifie chacune des classes représentées par le point de jointure `target` (voir ligne 2). Cette modification permet d'ajouter la super-classe `class` à l'ensemble des super-classes des classes ciblées par le point de jointure.

5.4.3 Fusion

Les adaptations de type `Fusion` sont des adaptations à la fois comportementales et fonctionnelles, car elles fusionnent des classes ou des méthodes. La fusion d'un ensemble de classes est une opération récursive car elle nécessite la fusion des méthodes. La fusion d'un ensemble de méthodes est une opération qui concatène (dans l'ordre du point de jointure `source`) les corps de méthode ayant une signature identique. La fusion nécessite donc deux points de jointure : `target` et `source`. Tous les éléments de source sont fusionnés dans chacun des éléments de `target`. En fonction du point de jointure `source` la fusion est une fusion de classes ou une fusion de méthodes. Les deux points de jointure doivent donc contenir tous deux le même type d'élément (des classes ou des méthodes).

```
1   Procedure Fusion.realizeAdaptation() {
2     if ( source.getMethods() is empty) {
3       realizeClassesFusion();
4     } else {
5       realizeMethodsFusion();
6     }
7   }
```

En fonction du type des cibles disponibles présentes dans `source`, la fusion est soit une fusion de méthodes `realizeMethodsFusion()` soit une fusion de classes `realizeClassesFusion()`.

```
1   Procedure Fusion.realizeClassesFusion () {
2     forall classtarget : Class in target.getClasses() {
3       forall c : Class in source.getClasses() {
4         fusion(c,classtarget);
5       }
6     }
7   }
```

La fusion de classes fusionne chaque classe présente dans `source` dans chacune des classes de `target`. L'ordre utilisé suit celui de l'ensemble `source`.

```
1   Procedure Fusion.realizeMethodsFusion () {
2     forall methodtarget : Method in target.getMethods() {
3       forall m : Method in source.getMethods() {
4         fusion(m,methodtarget);
5       }
6     }
7   }
```

De même, la fusion de méthodes fusionne chaque méthode présente dans `source` avec les méthodes présentent dans `target` et ayant une signature compatible.

```
1   Procedure Fusion.fusion (source, dest : Class) {
2     //Fusion on heritance's hierachy
3     forall c : Class in source.getSuperClasses() {
4       if ( c not in dest.getSuperClasses() ) {
5         dest.getSuperClasses().append(c);
6       }
7     }
8     //Method's fusion
9     forall m : Method in source.getMethods() {
10      Method oldMethod = dest.getMethod(m.getSignature());
11      if ( m != null ) {
12        fusion(m.clone(),oldMethod);
13      } else {
14        Method mDuplicata = m.clone();
15        mDuplicata.setClass(dest);
16        dest.getMethods().append(m.clone());
17      }
18    }
19    //Attribute's fusion
20    forall a : Attribute in source.getAttributes() {
21      Attribute oldAttribute = dest.getAttribute(a.getSignature());
22      if ( a == null )
23        Attribute aDuplicata = a.clone();
24        aDuplicata.setClass(dest);
25        dest.getAttributes().append(a.clone());
26      }
27    }
28  }
```

La fusion de deux classes est réalisée par la méthode `fusion(source,dest : Class)` de la classe `Fusion`. Cette fusion est effectuée en trois étapes :

– Les super-classes de la classe source sont ajoutées à la liste des super-classes de la classe cible (voir ligne 2 à 7).

– La fusion opère ensuite sur les méthodes (voir ligne 8 à 18) Pour chaque méthode de la classe source, si la méthode existe dans la classe de destination alors la fusion est déléguée à `fusion(source,dest : Method)`, sinon la méthode est ajoutée à la classe de destination.

– La dernière étape fusionne les attributs (voir ligne 19 à 27). Tous les attributs de la classe source non présents dans la classe de destination sont copiés dans cette dernière.

Nous décrivons maintenant l'opération de fusion unitaire de deux méthodes.

```
1   Procedure Fusion.fusion (source, dest : Method) {
2     Class c = dest.getClass();
3     //Hide original method
4     Method oldMethod = dest.clone();
5     oldMethod.setName( c.generateUniqueId( oldMethod.getName() ) );
6     oldMethod.setVisibilityModifier( new VisibilityModifier ( private ) );
7     c.getMethods().add(oldMethod);
8     //Copy and Hide source method
9     Method sourceClone = source.clone();
10    sourceClone.setName( c.generateUniqueId( sourceClone.getName() ) );
11    sourceClone.setVisibilityModifier( new VisibilityModifier ( private ) );
12    c.getMethods().add(sourceClone);
13    //Replace original method's body
14    String newBody = oldMethod.generateSourceCodeMethodCall();
15    newBody += sourceClone.generateSourceCodeMethodCall();
16    //Insert a call to oldMethod and sourceClone
17    dest.getBody().setInstruction(MethodBody.parseBody(newBody)) ;
18  }
```

La fusion de deux méthodes opère par masquage : la méthode source est copiée dans la classe de la méthode de destination et cette dernière est masquée (voir ligne 8 à 12), la méthode de destination est aussi masquée (voir ligne 3 à 7). Enfin, le corps de la méthode originale est modifié pour faire un appel à la méthode de destination masquée et ensuite un appel à la méthode source masquée elle-aussi. La fusion de deux méthodes crée donc une nouvelle méthode qui appelle séquentiellement les deux méthodes fusionnées.

5.5 BILAN

Ce chapitre montre comment à partir d'un ensemble de pré-requis sur le langage utilisé nous avons pu proposer un modèle quasi-indépendant du langage à étendre. Pour cela, un modèle du langage source MLS a été décrit en UML. MLS réifie toutes les entités qui sont nécessaires à notre modèle. Ces entités ont permis d'implémenter les adaptations mises en œuvre par notre modèle. Tout langage dont le métamodèle est inclus dans MLS est utilisable pour notre modèle. Or, comme MLS se contente d'utiliser des concepts classiques : la classe, la méthode, la variable, et le paquetage. Notre modèle est utilisable pour la plupart des langages à objets à classes.

Les nouvelles entités introduites par notre modèle ont été elles-aussi présentées et décrites en UML. Ces nouvelles entités sont : l'adaptateur, la variable d'adaptateur, le point de jointure, et l'adaptation. Elles mettent en oeuvre la composition des préoccupations dans notre modèle.

Enfin, toutes les adaptations supportées par notre modèle ont été implémentées en utilisant MLS et un pseudo-langage algorithmique.

Notre modèle répond au cahier des charges (voir chapitre 4) car il possède les points suivants :

– Une entité pour encapsuler les adaptations : l'`Adaptateur`. Ce dernier peut être abstrait, et il peut contenir des variables pour abstraire les points de jointure des adaptations. L'héritage entre adaptateur est aussi possible.

– Un adaptateur peut être soit *in-situ*, soit *ex-situ*. Ceci influence la sémantique des adaptations.

– Les variables peuvent être abstraites. L'héritage d'adaptateurs permet alors de les concrétiser. Ceci est aussi valable pour les adaptations.

– Le système de point de jointure permet de cibler : soit des classes, soit des méthodes, ou soit des variables.

– Les six adaptations nécessaires à la composition des préoccupations pour les langages à classes sont supportées par notre modèle.

Le chapitre suivant présente une implémentation du modèle que nous venons de décrire. Cette implémentation est ensuite évaluée (voir chapitre 7) sur les mêmes exemples que ceux utilisés lors

de l'état de l'art (voir chapitre 3). Ceci va permettre de confronter ce modèle à des cas d'utilisation réels.

Chapitre 6

Implémentation de JAdapt

Ce chapitre présente une implémentation de notre modèle. L'objectif principal de ce modèle est de définir une extension pour les langages à objets à classes qui permette de supporter une séparation des préoccupations. Cette dernière doit répondre à plusieurs impératifs qui ont été présentés dans la section 3.4.

L'implémentation de notre modèle correspond à un préprocesseur pour le langage à étendre. D'un point de vue technique, une implémentation de notre modèle pour un langage donné (noté *L*) se présente donc comme un programme (noté *T*) dont l'exécution est liée à l'environnement d'exécution et de compilation du langage *L*. Le but de *T* est d'étendre le langage *L* pour qu'il supporte la séparation des préoccupations (appelons *L*++ cette extension). *T* est une transformation de programme écrit en *L*++ vers *L* afin que celui-ci puisse être compilé et exécuté par les outils fournis avec *L*.

L'implémentation du modèle JAdapt s'apparente donc à un transformateur de programme ou encore à un compilateur de haut niveau. Comme la transformation de programme trouve aujourd'hui de nouveaux emplois dans l'approche MDA [MDA 04], il paraît important de présenter les choix et les techniques d'implémentation de notre modèle. Ces considérations ont une portée plus générale et peuvent servir de réflexion pour l'extension de langage à objets.

Nous allons d'abord discuter du langage utilisé pour écrire le code source des programmes qui vont être adaptés à l'aide de notre modèle. Ensuite nous évoquerons les différents outils existant dans l'état de l'art afin d'étudier l'apport engendré pour notre implémentation. Toutes ces considérations vont fixer le langage utilisé pour implémenter notre modèle, car ces outils ne sont pas tous utilisables avec tous les langages existants. Enfin, certains détails techniques de l'implémentation de JAdapt vont être présentés.

6.1 CHOIX DU LANGAGE A ETENDRE

Les approches de séparation des préoccupations étudiées dans le chapitre 2 sont des extensions de langages existants. Ces extensions introduisent la notion de séparation des préoccupations. Etendre l'existant permet toujours de réutiliser et de pérenniser des investissements. Nous nous plaçons résolument dans cette logique pour la mise en oeuvre de JAdapt et nous proposons de réaliser cette extension sous forme de transformation de modèle. Le modèle présenté dans le chapitre 5 est un PIM (c'est-à-dire un modèle indépendant de la plate-forme suivant la terminologie MDA) et que celui proposé ci-dessous sera un PSM (modèle dépendant de la plate-forme, toujours selon la terminologie MDA) pour ce modèle.

Notre modèle est utilisable pour les langages à objets à classes. Il est donc raisonnable de choisir parmi les langages suivants : C++, Java, Eiffel, Smalltalk ou C#.

La suite de ce chapitre va donc en particulier consister à montrer comment étendre le langage Java pour qu'il intègre les facilités offertes par notre modèle. Le choix du langage Java a été motivé par plusieurs raisons : *i)* Java est un langage disposant d'outils libres de droit, *ii)* Java est utilisable sur une grande variété de machines, *iii)* la plupart des extensions de langages pour la sépara-

tion des préoccupations concernent Java (par exemple : AspectJ/, Hyper/J, ComposeJ …) et peuvent donc nous servir de plate-forme d'implémentation.

Par ailleurs, Java utilise une machine virtuelle et donc nous pouvons intervenir à deux niveaux : soit au niveau du code source (le langage Java lui-même), soit au niveau du langage intermédiaire interprété par la machine virtuelle (le *byte-code Java*). Il faut donc choisir si JAdapt se placera au niveau du byte-code ou au niveau du code source et dans ce dernier cas, s'il effectuera les transformations de programme vers du code source ou directement vers du *byte-code*.

La nécessité d'offrir un accès simple pour le programmeur d'application, notamment pour faciliter la vérification des transformations lors du développement, nous a conduit à placer notre modèle au niveau du code source. Nous tenons cependant à préciser que ce ne sont pas des raisons techniques qui ont imposé ce choix puisque le *byte-code* Java conserve la structure hiérarchique des classes du programme. Le *byte-code* Java possèderait donc les informations suffisantes pour implémenter un transformateur de programme.

En ce qui concerne le code cible des transformations il n'y a aucun avantage à produire directement du *byte-code* plutôt que du code source ; les transformations se feront donc vers le code source Java. En effet, produire du *byte-code* serait à la fois plus complexe et surtout nécessiterait de faire évoluer la transformation en même temps que la machine virtuelle évolue elle-même. Par ailleurs, les compilateurs fournis avec les différentes machines virtuelles Java ne nécessitent aucune amélioration par rapport aux besoins de notre implémentation. Sans vouloir rentrer trop dans les détails techniques, disons simplement que la compilation « *just-in-time* » de Java permet de pallier à la plupart des optimisations qui auraient pu être réalisées directement si un compilateur prévu pour la séparation des préoccupations avait lui-même produit le *byte-code* Java.

Notons néanmoins que travailler sur le byte-code permet de réaliser certaines adaptations directement au *run-time*. L'état de l'art (et en particulier JAC) montre clairement qu'une approche de composition d'aspects dynamique (donc au *run-time*) permet une flexibilité accrue pour les préoccupations non fonctionnelles, en particulier le fait de pouvoir activer ou désactiver à la demande ces dernières.

Maintenant que le choix du langage a été expliqué, il semble opportun de s'intéresser aux outils pour transformer des programmes d'un code source Java étendu vers du code source Java natif. L'inventaire que nous proposons n'est pas exhaustif mais il est suffisant pour donner au lecteur une bonne idée de nos besoins.

6.2 OUTILS DE TRANSFORMATION DE PROGRAMME

Nous avons montré que JAdapt est un cas particulier de transformation de programme. Il est donc utile, avant d'aborder les différents outils, de rappeler brièvement le cadre général de la transformation de programme afin de mettre en évidence les spécificités de notre approche et de pouvoir proposer un éventail d'outils qui pourraient être utilisés pour implémenter notre modèle.

6.2.1 Généralités sur la transformation de programme

La transformation de programme permet d'appliquer un ensemble de transformations à un programme pour produire un autre programme (voir figure 17).

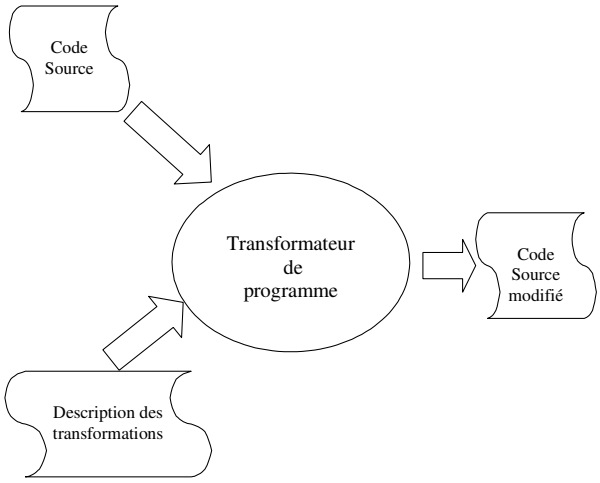


Figure 17. *Schéma d'un transformateur de programme*

La transformation de programme nécessite en premier lieu des analyseurs (syntaxiques et lexicaux) pour extraire une représentation logique du code source. Le code source doit être analysé et transformé (*parsing*) sous forme d'arbre de syntaxe abstraite (*AST)* pour permettre son traitement de manière informatique.

Les analyseurs syntaxiques et lexicaux sont les principaux outils utilisés pour faire de la transformation de programme. Ces outils existent pour quasiment tous les langages ; des grammaires décrivant la plupart des langages sont très souvent fournies avec les analyseurs.

Nous allons maintenant voir les spécificités de notre approche.

6.2.2 Transformation de programme réalisée à partir de notre modèle

Notre modèle effectue des transformations de programme pour composer les préoccupations, le cadre est le même que celui décrit par [FS 98]. Plus précisément, les opérations d'adaptation déclenchent des transformations sur le programme originel pour composer les préoccupations. La transformation de programme dans le cadre de notre modèle est donc dirigée par les opérations d'adaptation dont la liste a été rappelée dans la section 5.3.4.

Les adaptations utilisées dans notre modèle induisent des transformations de programme au niveau des classes. Les transformations supportées par notre modèle et appliquées au langage Java sont : la modification des liens d'héritage de la hiérarchie de classes ou d'interfaces, la modification du paquetage (package en Java) de la classe, l'ajout d'une nouvelle méthode ou d'une nouvelle variable d'instance ou de classe, la modification du code d'une méthode.

Ces transformations de programme nécessitent un arbre de syntaxe abstraite réifiant la hiérarchie de classes du programme original. Cet arbre de syntaxe doit avoir l'instruction comme mesure de granularité pour permettre la modification du code des méthodes.

Notons la possibilité de travailler avec la méthode comme mesure de granularité en ne perdant qu'une petite partie de l'expressivité des points de jointures utilisés par les adaptation. Cette perte d'expressivité peut néanmoins conduire au problème des « *jumping aspects* » [BMV 00].

Comme cela a été dit plus haut, le langage sur lequel nous proposons d'appliquer l'extension décrite par notre modèle est Java. Nous disposons par conséquent d'un large éventail d'outils. Nous présentons ci-après ceux que nous avons jugés intéressants pour mettre en œuvre notre modèle.

JAdapt est une transformation de programme produisant du code source Java. Cette production est dirigée par des adaptations décrites avec le formalisme de notre modèle adapté au langage Java. La production utilise du code source Java pour la description de l'application et donc des préoccupations.

L'ensemble des outils qui est présenté est loin d'être exhaustif, mais il donne un aperçu des différentes catégories d'outil réalisant des transformations de programme à partir d'un code source Java et d'instructions d'adaptation. On peut noter par ailleurs que la plupart de ces outils sont écrits en Java.

6.2.2.1 AspectJ

AspectJ [ASP 03] est une extension de Java permettant de faire de la séparation des préoccupations avec le modèle des aspects [KLM 97]. Cet outil et le modèle associé ont déjà été présentés dans les sections 2.3.4 et 2.4.1 et ils ont été étudiés plus en détail dans le chapitre 3.

Les chapitres précédents montrent qu'AspectJ fait partie des modèles sur lesquels nous nous sommes appuyés pour définir les spécifications de notre modèle (voir 3.4). La principale différence entre AspectJ et JAdapt est l'expressivité du langage utilisé pour décrire les adaptations nécessaires à la composition des préoccupations.

AspectJ est une extension de Java qui permet de décrire des traitements dans les aspects et de les composer avec les classes d'un programme. De son côté, JAdapt a pour vocation de décrire des adaptations et de les composer. On peut dire que décrire des aspects ou des adaptations correspond à deux manières différentes de décrire des transformations de programme.

De même, comme cela a été dit plus haut, JAdapt et AspectJ opèrent tous les deux sur du code source Java pour produire aussi du code source Java. Il est donc naturel de penser qu'AspectJ peut être utilisé comme moteur de transformation de programme pour JAdapt.

De plus, s'il est possible de proposer un modèle de transformation entre le formalisme que nous utilisons pour décrire la composition et celui d'AspectJ, alors il est possible d'implémenter le mécanisme de composition de JAdapt avec AspectJ. Ceci revient à dire que pour pouvoir utiliser AspectJ comme outil pour implanter JAdapt, il faut montrer qu'il existe une transformation de notre formalisme vers celui proposé par AspectJ.

Or, on constate que cette transformation nécessite : *i)* d'avoir accès à des informations contenues dans le code source comme les relations d'héritage entre les classes ou le corps des méthodes *ii)* d'effectuer des vérifications sur le code source ; il s'agit principalement des vérifications sur l'existence de classe et sur les signatures des méthodes.

AspectJ ne peut pas nous fournir ces informations car même s'il travaille en interne sur un arbre de syntaxe abstraite représentant le code source et contenant toutes ces informations, il n'est pas possible d'accéder à cet arbre

Néanmoins, aux vues des similitudes entre les deux modèles, une transformation de notre formalisme vers le formalisme des aspects nous paraît envisageable. On constate donc qu'AspectJ pourrait être un très bon candidat pour servir d'outil de mise en œuvre de notre modèle. Cependant nous avons choisi de ne pas le sélectionner pour plusieurs raisons :

– AspectJ ne permet pas l'accès à l'arbre de syntaxe abstraite, ce qui conduit à effectuer des traitements redondants,

– Utiliser AspectJ pour implémenter JAdapt implique d'accepter que la transformation soit dépendante de la syntaxe utilisée dans les deux outils. Or, AspectJ est un produit en constante évolution et sa syntaxe concrète a déjà subi des changements (d'autres sont sûrement à venir). Ces évolutions syntaxiques conduiraient à un important travail de maintenance sur le modèle de transformation. S'il fournissait, par exemple, une deuxième syntaxe basée sur XML, la pérennité de la transformation serait assurée (car même si la syntaxe change il est facilement envisageable de prendre en compte ces changements par une transformation XML vers XML).

6.2.2.2 Hyper/J

Hyper/J [HYP 03] est une extension de Java permettant de faire de la séparation des préoccupations avec le modèle des sujets [HO 93]. Cet outil et le modèle associé ont déjà été présentés dans les sections 2.3.5 et 2.4.2. De plus, une étude détaillée de ses possibilités a été proposée dans le chapitre 3.

Bien que le modèle des sujets diffère des aspects, Hyper/J peut – dans notre contexte d'utilisation – se substituer à AspectJ. La similitude des deux outils ne s'arrête pas là. En effet, les critiques réalisées à l'encontre d'AspectJ relatives à son utilisation pour implanter notre modèle s'appliquent aussi à Hyper/J. Cela nous conduit bien évidemment à ne pas le choisir ; il en va de même (pour les mêmes raisons), des autres outils présentés dans le chapitre 2.

6.2.2.3 Eclipse

Eclipse [ECL 04] est entre autre un environnement intégré de développement pour Java. Face à un grand nombre de produits similaires, Eclipse a retenu notre attention pour les raisons suivantes : *i)* il est mis a jour régulièrement, *ii)* il repose entièrement sur un système de plug-in qui assure son ouverture, *iii)* il est très bien documenté, *iv)* le code source est disponible, *v)* il est entièrement programmé en Java.

Le plug-in fourni par défaut par Eclipse pour faire du Java contient des outils qu'il est possible de réutiliser. Parmi ces outils notons la présence d'un analyseur lexical pour du code source Java. Cet analyseur lexical produit des arbres de syntaxe abstraite et offre une API pour les manipuler et les retransformer en code source Java. Nous disposons donc de toutes les fonctionnalités nécessaires pour effectuer des transformations de programme et donc implémenter notre modèle

L'ouverture de la plate-forme Eclipse et de ses plug-ins (en particulier le plug-in Java) permettent d'adapter le fonctionnement Eclipse pour prendre directement en compte notre modèle de séparation des préoccupations. Cela permet donc aux utilisateurs d'Eclipse d'utiliser de manière transparente JAdapt car celui-ci serait alors intégré à la plate-forme.

Implémenter notre modèle comme un plug-in pour Eclipse est une approche intéressante pour faciliter le développement de JAdapt grâce aux API et plug-ins fournis avec Eclipse. Cela permet aussi de rendre directement utilisable notre modèle pour tous les utilisateurs d'Eclipse.

Après avoir étudié les apports des environnements de programmation intégrés nous évaluons l'utilisation de la métaprogrammation pour implémenter notre modèle.

6.2.2.4 Métaprogrammation

L'apport de la métaprogrammation pour la séparation des préoccupations a été présenté dans la section 2.3.2 et étudiée de manière plus détaillée dans le chapitre 3. Cette approche peut être utilisée comme un outil pour développer des approches de plus haut niveau d'utilisation comme le montrent [Duc 99] et [Bou 00].

Un des objectifs de la métaprogrammation est d'ouvrir les langages afin de modifier leur comportement, cette ouverture peut être utilisée pour implémenter notre approche. En utilisant les pos-

sibilités de modification des classes offertes par le niveau méta il nous est possible d'implémenter notre modèle (pour un exemple voir [Bou 00]).

Par contre, face à la complexité de la méta programmation et au fait que nous n'avons pas besoin de tous les avantages qu'elle apporte, il nous semble raisonnable de choisir une approche plus simple et surtout plus adaptée à nos problèmes.

Dans cette optique nous allons maintenant nous pencher sur l'utilisation d'outils plus traditionnels comme les analyseurs syntaxiques et lexicaux.

6.2.2.5 Outils traditionnels dédiés aux compilateurs

La transformation de programme nécessite une réification modifiable de l'entité à transformer (code source, modèle …). Comme cela a déjà été dit, JAdapt nécessite une réification modifiable du code source Java. Les outils qui permettent de transformer du code source en un arbre de syntaxe abstraite sont des outils largement utilisés pour construire des compilateurs.

Pour le langage Java les principaux outils disponibles sont ANTLR [ANT 04] et Javacc [JCC 04]. Ce sont deux outils qui fournissent un framework permettant de construire des analyseurs syntaxiques et lexicaux, des compilateurs et des transformateurs à partir d'une grammaire.

Aussi bien ANTLR que Javacc sont fournis avec un ensemble de grammaires pour les langages fréquemment utilisés : SQL, HTML, C, … Par contre seul Javacc propose une grammaire supportant le langage Java y compris celle de la version 1.5.

Compte tenu des évolutions du langage Java qui restent fréquentes surtout par rapport à d'autres langages comme par exemple C++, il serait préférable d'opter pour l'utilisation de Javacc qui permet à l'heure actuelle de pouvoir prendre en compte Java jusqu'à la version 1.5

Ces outils prennent en charge la construction d'un arbre de syntaxe abstraite modifiable à partir du code source Java. Ils sont cependant relativement pauvres en terme de fonctionnalités par rapport aux environnements de développement intégrés. Ainsi, par rapport à Eclipse équipé du plug-in pour Java, ANTLR et JavaCC n'apportent rien de plus pour la transformation de programmes écrits en Java.

6.2.2.6 SmartTools

SmartTools [ACD 01] est un environnement de développement qui fournit une édition structurée et des outils sémantiques. Un des objectifs de SmartTools est de générer tous les outils nécessaires à l'implémentation des langages (métiers ou non).

SmartTools pourrait donc servir comme outil pour générer la partie de notre implémentation dont la fonction est de construire des arbres de syntaxes abstraites représentant le code source du programme et le code source de la composition.

Néanmoins, SmartTools nécessite plus de travail par rapport à Eclipse car ce dernier fournit un plug-in pour Java qui réalise déjà le travail de construction d'arbres de syntaxe abstraite à partir de code source Java. Eclipse fournit aussi un environnement de développement très complet et très ouvert.

SmartTools est donc envisageable uniquement pour générer les outils nécessaires à notre langage métier. Mais, comme nous allons le voir par la suite dans la section 6.3.1, nous avons opté pour gagner du temps sur le fait d'utiliser des fichiers au format XML contenant le code de la composition.

6.2.3 Conclusion

L'éventail d'outils que nous avons présenté n'est sûrement pas exhaustif mais il est suffisamment représentatif des grandes catégories d'outil disponible.

Parmi ces grandes catégories d'outil disponible, nous avons commencé par étudier l'utilisation de langages pour la séparation des préoccupations comme AspectJ ou Hyper/J. Nous avons constaté le manque d'ouverture des outils fournis avec ces langages et nous n'en avons retenu aucun.

Ensuite nous avons porté notre attention sur les environnements de développement intégrés, et plus particulièrement sur Eclipse. L'ouverture de cette plate-forme est maximale grâce à un système de plug-in. Eclipse dispose de tous les outils nécessaires à l'implémentation de notre approche et lui offre en supplément un environnement *ad hoc*.

La métaprogrammation, les outils dédiés à la construction de compilateur, et SmartTools n'apportent rien par rapport à l'utilisation d'un environnement comme Eclipse.

Nous avons donc choisi d'implémenter notre modèle sous la forme d'un *plug-in* pour Eclipse. Maintenant que le cadre de l'implémentation et de l'utilisation de notre modèle est fixé nous décrirons notre implémentation.

6.3 PRESENTATION DE L'IMPLEMENTATION

Cette section présente l'architecture de JAdapt, une implémentation pour Java de notre modèle de séparation et de composition de préoccupations qui a été présenté dans le chapitre 5. Il est réalisé avec Eclipse qui joue le rôle de plate-forme d'accueil pour JAdapt.

Notre implémentation est réalisée comme une extension du plug-in Java pour Eclipse. En effet, ce plug-in est un plug-in composite (au sens de [GHJ 99]) qui s'intègre à l'environnement Eclipse de manière à lui permettre de supporter le langage Java comme langage de programmation.

Ce plug-in Java est fourni en standard dans la distribution et les fonctionnalités suivantes vont être particulièrement utiles : *i)* des analyseurs syntaxiques et lexicaux pour le langage Java, *ii)* un arbre de syntaxe abstraite modifiable qui est produit par *i* et peut à nouveau être transformé en code Java, *iii)* un compilateur Java complet, *iv)* une nature[1] de projet Eclipse qui est faite pour le langage Java.

Pour mettre en œuvre notre plug-in tout en respectant l'esprit de l'implémentation de la plate-forme, il est nécessaire d'étendre la nature du *projet Java*. Cette nature contient, entre autre, la liste des actions nécessaires à la compilation d'un projet Java. L'intégration de notre plug-in dans la plate-forme se fait en modifiant cette liste d'actions afin d'y insérer notre propre précompilateur avant l'appel au compilateur Java d'Eclipse. Ce précompilateur effectue les adaptations décrites par l'utilisateur pour composer les préoccupations.

Les différentes sections suivantes présentent les différentes fonctionnalités de notre précompilateur. La première sous-section présente la partie avant qui est responsable de l'extraction des informations nécessaires aux autres parties pour fonctionner. La deuxième sous-section présente le cœur de JAdapt qui est responsable de la composition des préoccupations. Enfin, la troisième sous-section retransforme le résultat de la partie centrale en code source Java afin de le faire compiler par un compilateur de code source Java.

[1] La nature d'un projet dans la plate-forme Eclipse permet de faire correspondre le langage utilisé pour implémenter le projet et les différents plug-ins utilisés par la plate-forme.

6.3.1 Extraction de l'information

Notre modèle permet de composer des préoccupations, nous allons voir dans cette partie quelle est l'information à extraire du code source pour composer les préoccupations.

Pour composer des préoccupations il faut : *i)* une description des préoccupations, *ii)* une description de la composition à effectuer. La précision de ces deux informations dépend du modèle de composition des préoccupations et plus particulièrement de son expressivité.

Le fait de créer une nouvelle nature de projet permet aussi de supporter la définition des adaptations à réaliser par notre modèle pour composer les préoccupations car la nouvelle nature s'utilise conjointement avec la nature Java d'origine d'Eclipse pour prendre en compte les fichiers correspondant à l'implémentation des adaptateurs. Nous rappelons à ce sujet que notre modèle permet d'encapsuler les adaptations dans des hiérarchies d'Adaptateur.

Nous avons choisi d'utiliser le langage XML [XML 04] pour décrire nos adaptateurs, pour plusieurs raisons : *i)* nous avons pu définir une syntaxe claire de nos données XML grâce à l'utilisation de Schéma XML [SCH 04], *ii)* Eclipse possède un éditeur de fichier XML, *iii)* cette combinaison nous a permis d'utiliser l'API JAXB [JAX 04] qui permet de générer un ensemble de classes Java représentant l'information contenue dans un fichier XML. Ce choix engendre un gain de temps pour extraire et utiliser l'information décrivant nos adaptateurs et il permet toujours de prendre en compte plus tard une deuxième syntaxe plus concrète et plus facilement utilisable par le programmeur.

Ci-dessous le Schéma XML permettant de décrire les fichiers XML contenant des adaptateurs.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3       <xsd:element name="jadapt" type="jadaptType"/>
4       <xsd:element name="comment" type="xsd:string"/>
5       <!-- ROOT NODE = JAdapt -->
6       <xsd:complexType name="jadaptType">
7           <xsd:sequence>
8               <xsd:element maxOccurs="1" minOccurs="1" name="package" type="xsd:string"/>
9               <xsd:element maxOccurs="unbounded" minOccurs="0" name="import"
10                           type="xsd:string"/>
11              <xsd:element maxOccurs="1" minOccurs="1" name="adapter" type="adapter"/>
12          </xsd:sequence>
13      </xsd:complexType>
14      <!-- Adaptater -->
15      <xsd:complexType name="adapter">
16          <xsd:sequence>
17              <xsd:choice maxOccurs="unbounded" minOccurs="0">
18                  <xsd:element maxOccurs="1" minOccurs="0" name="variable"
19                               type="variable"/>
20                  <xsd:element maxOccurs="1" minOccurs="0" ref="adaptation"/>
21              </xsd:choice>
22          </xsd:sequence>
23          <xsd:attribute name="name" type="xsd:string" use="required"/>
24          <xsd:attribute name="extends" type="xsd:string" use="optional"/>
25          <xsd:attribute name="abstract" type="xsd:boolean" use="optional"/>
26      </xsd:complexType>
```

Un fichier xml contient : un package, des directives importations et un adaptateur.

Un adaptateur contient des variables et des adaptations. Il possède un nom, un super-adaptateur et peut être abstrait

```
<!-- Variable -->
<xsd:complexType name="variable">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="abstract" type="xsd:boolean" use="optional"/>
            <xsd:attribute name="comment" type="xsd:string" use="optional"/>
            <xsd:attribute name="type" use="optional">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="Method"/>
                        <xsd:enumeration value="Class"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

Une variable possède un nom, un type, un commentaire, et une éventuelle valeur.

```
<!-- Adaptation (base type) -->
<xsd:complexType name="adaptationType">
    <xsd:attribute name="name" type="xsd:string" use="optional"/>
    <xsd:attribute name="abstract" type="xsd:boolean" use="optional"/>
    <xsd:attribute name="comment" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:element name="adaptation" type="adaptationType"/>
<xsd:element name="After" substitutionGroup="adaptation" type="AfterType"/>
<xsd:element name="Before" substitutionGroup="adaptation" type="BeforeType"/>
<xsd:element name="Around" substitutionGroup="adaptation" type="AroundType"/>
<xsd:element name="OnException" substitutionGroup="adaptation"
             type="OnExceptionType"/>
<xsd:element name="implement" substitutionGroup="adaptation" type="implementType"/>
<xsd:element name="introduce" substitutionGroup="adaptation" type="introduceType"/>
<xsd:element name="fusion" substitutionGroup="adaptation" type="fusionType"/>
```

Une adaptation possède un nom, un commentaire et peut être abstraite

```
<!-- Introduce -->
<xsd:complexType name="introduceType">
    <xsd:complexContent>
        <xsd:extension base="adaptationType">
            <xsd:choice>
                <xsd:sequence>
                    <xsd:element name="pointcut" type="pointcut"/>
                    <xsd:choice>
                        <xsd:element name="method" type="method"/>
                        <xsd:element name="classvariable" type="classvariable"/>
                    </xsd:choice>
                </xsd:sequence>
            </xsd:choice>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Cette adaptation permet d'introduire soit une variable, soit une méthode dans un ensemble de classe cible.

```
<!-- After,Before,Around,OnException -->
<xsd:complexType name="InterceptionType">
    <xsd:complexContent>
        <xsd:extension base="adaptationType"/>
    </xsd:complexContent>
</xsd:complexType>
<!-- Interception -->
<xsd:group name="Interception">
    <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1" name="pointcut" type="pointcut"/>
        <xsd:element maxOccurs="1" minOccurs="1" name="do" type="do"/>
    </xsd:sequence>
</xsd:group>
```

Super-classe de toutes les interceptions.

Définit le contenu commun à toutes les interceptions.

```
 <xsd:complexType name="AfterType">
    <xsd:complexContent>
        <xsd:extension base="InterceptionType">
            <xsd:sequence>
                <xsd:group ref="Interception"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="BeforeType">
    <xsd:complexContent>
        <xsd:extension base="InterceptionType">
            <xsd:sequence>
                <xsd:group ref="Interception"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="AroundType">
    <xsd:complexContent>
        <xsd:extension base="InterceptionType">
            <xsd:sequence>
                <xsd:group ref="Interception"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="OnExceptionType">
    <xsd:complexContent>
        <xsd:extension base="InterceptionType">
            <xsd:sequence>
                <xsd:group ref="Interception"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Quatre types d'interceptions disponibles.

```
 <!-- PointCut -->
 <xsd:complexType name="pointcut">
    <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="0" name="packages"
                     type="xsd:string"/>
        <xsd:element maxOccurs="1" minOccurs="1" name="classes" type="xsd:string"/>
        <xsd:element maxOccurs="1" minOccurs="0" name="methods" type="xsd:string"/>
        <xsd:element maxOccurs="1" minOccurs="0" name="parameters"
                     type="xsd:string"/>
    </xsd:sequence>
 </xsd:complexType>
```

Définition du point de jointure en vue éclatée pour faciliter son traitement.

```
 <!-- Do -->
 <xsd:complexType name="do">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string"/>
    </xsd:simpleContent>
 </xsd:complexType>
```

Le `Do` des interceptions est une simple chaîne de caractères qui contient du code source.

```
 <!-- Method -->
 <xsd:simpleType name="MethodParameters">
    <xsd:list itemType="xsd:string"/>
 </xsd:simpleType>
 <xsd:complexType name="method">
    <xsd:sequence>
        <xsd:element maxOccurs="1" minOccurs="1" name="name" type="xsd:string"/>
        <xsd:element maxOccurs="1" minOccurs="0" name="parameters"
                     type="MethodParameters"/>
        <xsd:element maxOccurs="1" minOccurs="1" name="body" type="xsd:string"/>
    </xsd:sequence>
 </xsd:complexType>
```

Définition d'une méthode et de ces paramètres. Cette définition est utilisée pour introduire une nouvelle méthode dans des classes.

```
 <!-- Variable  -->
 <xsd:complexType name="classvariable">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="type" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:simpleContent>
 </xsd:complexType>
```

Définition d'une variable d'une classe. Utilisée par l'adaptation de type introduction.

```
165     <!-- Implements -->
166     <xsd:complexType name="implementType">
167         <xsd:complexContent>
168             <xsd:extension base="adaptationType">
169                 <xsd:sequence>
170                     <xsd:element name="interface" type="xsd:string"/>
171                     <xsd:element name="in" type="xsd:string"/>
172                 </xsd:sequence>
173             </xsd:extension>
174         </xsd:complexContent>
175     </xsd:complexType>
176     <!-- fusion -->
177     <xsd:complexType name="fusionType">
178         <xsd:complexContent>
179             <xsd:extension base="adaptationType">
180                 <xsd:sequence>
181                     <xsd:element name="target" type="pointcut"/>
182                     <xsd:element name="source" type="pointcut"/>
183                 </xsd:sequence>
184             </xsd:extension>
185         </xsd:complexContent>
186     </xsd:complexType>
187 </xsd:schema>
```

Interception permettant d'introduire une interface dans un ensemble de classes.

Interception de type fusion. Elle possède deux points de jointures.

Nous, n'allons pas détailler ce Schéma XML, il est juste donnée ici à titre d'information. Notons seulement qu'il contient une réification dans le formalisme [SCH 04] de toutes les entités introduites par notre modèle (voir section 5.3). Certaines concessions ont été faites par rapport au modèle présenté dans le chapitre précédent. Principalement à cause de JAXB [JAX 04] qui ne supporte par toutes les fonctionnalités des Schémas XML, Mais aussi ponctuellement pour simplifier notre implémentation de JAdapt.

La partie avant de notre précompilateur extrait donc l'information nécessaire à la composition à partir du code source d'un projet et des fichiers XML qui contiennent une description des adaptateurs. Nous allons voir maintenant comment cette information est utilisée.

6.3.2 Traitement de l'information : composition des préoccupations

JAdapt encapsule les préoccupations dans des paquetages Java et les adaptateurs dans des fichiers XML. JAdapt permet de composer les préoccupations grâce aux adaptations contenues dans les adaptateurs. Toutes ces informations ont été extraites et réifiées par la partie avant ; elles sont donc pleinement manipulables.

Nous allons maintenant présenter les différentes sous-parties de la composition des préoccupations. Dans un premier temps, les adaptateurs sont contrôlés par un «vérificateur de typage » puis, la hiérarchie des adaptateurs est aplatie et, enfin, les adaptations contenues dans les adaptateurs sont réalisées.

6.3.2.1 Vérification du typage des adaptateurs

La vérification du typage des adaptateurs correspond à deux phases qui s'enchaînent. La première est un filtre qui vérifie certaines propriétés sur les adaptateurs qui sont nécessaires à la phase suivante. La deuxième phase fait une vérification plus poussée de ces mêmes adaptateurs. Si cette deuxième phase ne produit pas un résultat globalement correct alors la suite de l'exécution est court-circuitée. En effet, l'étape suivante ne peut être convenablement réalisée s'il subsiste des erreurs de typage.

La première phase de la vérification du typage est composée d'un ensemble de tests effectués dans l'ordre suivant :

1- **Respect des conventions de nommage**. Nous avons choisi d'appliquer les conventions de nommage de Java : le nom de l'adaptateur doit correspondre au nom du fichier qui le contient (sans l'extension), de plus ce nom doit être unique pour un même paquetage Java.
2- **Unicité des identificateurs dans les adaptateurs**. Les noms de variable et les noms des adaptations contenus dans un adaptateur doivent être uniques dans l'espace de nommage de l'adaptateur.
3- **Adaptateur générateur**. Les adaptateurs peuvent générer un nouveau paquetage et un paquetage ne peut être généré que par un seul adaptateur.

Tous les adaptateurs qui ne satisfont pas aux tests de la première phase ne sont plus considérés pour la suite ; ils sont rejetés (et les raisons sont signalées à l'utilisateur) et n'entrent pas en compte pour la composition des préoccupations. Les autres subiront les tests de la seconde phase. Celle-ci contient les tests suivants :

1- **Existence des importations**. De même qu'une classe Java un adaptateur peut importer des paquetages ou des classes. Ceux-ci doivent soit exister, soit être générés par un autre adaptateur.
2- **Caractère abstrait ou concret de l'adaptateur**. Tout comme les classes, les adaptateurs peuvent être abstraits ou concrets. Les cas suivants engendrent une erreur de typage :
   – Une variable abstraite présente dans un adaptateur concret ;
   – Une variable abstraite possède une valeur ;
   – Une variable est notée concrète mais ne possède pas de valeur ;
   – Une adaptation abstraite dans un adaptateur concret ;
   – Une adaptation abstraite possède toute l'information pour être réalisable ;
   – Une adaptation concrète ne possède pas toute l'information pour être réalisable.
3- **Existence d'un super-adaptateur**. Comme pour les classes les adaptateurs autorisent la déclaration d'une relation d'héritage. Le super-adaptateur d'un adaptateur doit exister dans l'espace de nommage construit en utilisant les conventions de visibilité des classes Java.

Si l'ensemble des adaptateurs admis à passer les tests de la seconde phase les satisfont tous, alors la vérification du typage est terminée. JAdapt passe à l'étape suivante qui est la résolution de l'héritage. Si le moindre adaptateur échoue, même à un seul test, alors la vérification du typage n'est pas correcte et le préprocesseur termine son exécution.

6.3.2.2 Aplatissement de la hiérarchie des adaptateurs

Notre modèle permet de définir des hiérarchies d'adaptateurs pour permettre d'abstraire et de réutiliser la composition des préoccupations à travers la définition et l'utilisation de protocoles de composition (voir chapitre 4 et chapitre 5).

Pour simplifier la composition des préoccupations qui est effectuée dans une troisième étape, on propose d'optimiser la hiérarchie des adaptateurs. La transformation qui est réalisée dans ce but permet de ne garder que les adaptateurs feuilles marqués comme étant non abstraits. Le lien d'héritage entre deux adaptateurs est alors supprimé et le super-adaptateur est inséré dans l'adaptateur. Cette opération correspond à l'aplatissement du lien d'héritage.

Les adaptateurs abstraits qui sont des feuilles de l'arbre d'héritage sont naturellement éliminés car le fait qu'ils soient abstraits ne leur permet pas de participer à la composition.

L'aplatissement d'un arbre d'héritage est une opération récursive qui part des feuilles de l'arbre d'héritage pour remonter à travers les relations d'héritage successives jusqu'à la racine. L'algorithme utilisé est le suivant :

```
1   Pour chaque a : Adaptateur ∈ Ensemble des feuilles faire
2     MettreAPlat_Recursif(a)
3   FinPour
4   Enlever de l'ensemble des feuilles les adaptateurs abstraits
5
6   procedure MetteAPlatRecursif(a : Adaptateur)
7   debut
8     Si a n'est pas mis à plat Et a possède un super adaptateur Alors
9
10      MetteAPlatRecursif(Super Adaptateur de a);
11
12      MettreAPlat(a);
13    FinSi
14  fin
```

L'aplatissement d'un adaptateur est réalisé par les opérations suivantes :

– Fusion des déclarations d'importation du super-adaptateur dans l'adaptateur.

– Fusion des variables et des adaptations du super-adaptateur dans l'adaptateur.

Ensuite, tous les adaptateurs abstraits sont retirés de l'ensemble des feuilles, et ce dernier ainsi réduit est transmis à l'étape suivante qui a la responsabilité de mettre en œuvre toutes les adaptations contenues dans ces adaptateurs.

6.3.2.3 Réalisation des adaptations

La réalisation des adaptations modifie l'arbre de syntaxe abstraite, qui a été construit par les analyses syntaxique et lexicale des fichiers sources du projet ; ces modifications dépendent du contenu des adaptateurs une fois aplatis.

Dans un premier temps les adaptateurs sont triés en fonction d'une relation de dépendance : un adaptateur *X* est dépendant d'un adaptateur *Y*, si *Y* génère un *paquetage* qui est importé par *X*. Les éventuels cycles dans le graphe des dépendances sont détectés pendant ce tri, et les adaptateurs incriminés sont rejetés.

Dans un deuxième temps, la liste triée des adaptateurs est parcourue, et les adaptations qu'ils définissent sont réalisées dans l'ordre de leurs déclarations.

Nous étudions maintenant comment l'arbre de syntaxe d'un programme doit être modifié pour réaliser tous les types d'adaptation supportés par notre modèle. Pour plus de détails sur l'implémentation des différentes adaptations, se référer à la section 5.4.

Notons que, quelle que soit d'adaptation à réaliser elle utilise au minimum un point de jointure. Les points de jointure (et les variables des adaptateurs) sont décrits par des chaînes de caractères qui contiennent des expressions régulières au format de l'API `java.util.regex` qui est fournie avec le langage Java. Cette API nous permet donc à partir du point de jointure de trouver toutes les cibles qui lui correspondent.

6.3.2.4 Réalisation d'une interception de méthode

Notre modèle propose quatre sortes d'interception de méthode :

– les interceptions avant la méthode,
– les interceptions après la méthode,
– les interceptions sur levée d'exception,
– les interceptions autour de la méthode.

Quel que soit le type d'interception une technique générale consiste à successivement : *i)* masquer la méthode originale qui est interceptée en remplaçant son nom par un nom unique généré automatiquement, *ii)* créer une méthode avec le nom et la signature de la méthode originale, et *iii)* créer le corps de cette méthode en fonction de l'interception demandée.

Les adaptations à appliquer lorsqu'il s'agit de l'interception d'accès aux variables de classe ou d'instance sont similaires à celles mises en œuvre pour l'interception de méthode. La principale différence consiste, pour masquer la variable, à remplacer tous les accès à cette variable par des appels de méthodes (qui sont à créer).

6.3.2.5 Réalisation d'une fusion de classes

La fusion de classes est une opération qui, comme son nom l'indique, fusionne un ensemble ordonné de classes dans chacune des classes cibles. Pour plus de détails sur la fusion, voir la section 5.4.3.

6.3.2.6 Ajout de membres ou d'interfaces à une classe

L'ajout de membres (variables ou méthodes) ou de nouvelles relations d'implémentation d'interface à une classe existante est une opération très simple car les modifications sur l'arbre de syntaxe sont élémentaires. Il faut simplement vérifier que le membre à ajouter n'existe pas déjà ; dans le cas contraire il faut refuser l'ajout sauf pour les méthodes. On considère alors l'ajout comme une redéfinition, ce qui revient à remplacer l'implémentation initiale par la nouvelle.

6.3.3 Partie arrière

La section 6.3.2 a effectué les modifications nécessaires à la composition des préoccupations directement sur l'arbre de syntaxe du programme.

La partie arrière projette l'arbre de syntaxe abstraite vers du code source Java. Celui-ci est ensuite compilé normalement par la plate-forme Eclipse grâce à la Nature Java sous-jacente.

Pour des raisons de déverminage, la partie arrière peut sauvegarder l'arbre de syntaxe qui contient les adaptations sous forme de fichier contenant du code source Java pour voir l'effet des adaptations sur le code source. Afin de ne pas perdre le code source initial, les classes modifiées par notre plug-in sont sauvegardées dans des fichiers dont les noms sont préfixés.

Nous proposons maintenant de faire le point sur les choix d'implémentation de notre modèle.

6.4 BILAN

L'implémentation de notre modèle correspond à l'implémentation d'un transformateur de programme ou à un compilateur de haut niveau. Nous avons choisi d'utiliser le langage Java pour

décrire le source des programmes à transformer car il offre plusieurs avantages : *i)* il est implanté par une machine virtuelle, *ii)* il possède une API riche, et *iii)* il existe de nombreux outils pour ce langage.

Nous avons présenté les différents outils disponibles pour faire de la transformation de programme dans le but de minimiser l'effort d'implémentation et de maintenance relatifs à notre modèle. Nous avons choisi d'utiliser la plate-forme Eclipse [ECL 04] car c'est un environnement de développement basé sur le concept de plug-in qui garantit son extensibilité.

Nous avons proposé dans ce chapitre une implémentation de notre modèle appelée JAdapt. Elle s'applique au langage Java et elle est réalisée sous la forme d'un plug-in pour Eclipse. Cette implémentation répond à plusieurs impératifs qui ont été présentés dans la partie 3.4. Elle supporte l'abstraction et l'héritage des adaptateurs et aussi toutes les formes d'adaptations que nous avons évoquées.

Il est intéressant de constater, pour finir, que la technique d'implémentation utilisée peut servir de réflexion pour l'élaboration d'outils comme les compilateurs de haut niveau. En effet, nous avons montré comment étendre une plate-forme de développement pour y intégrer une extension d'un langage existant ou faire de la transformation de programme. Nous pensons que les plates-formes de développement représente une avancée incontournable pour le développement d'application, il est donc important que ces dernières soit ouvertes pour pouvoir supporter des extensions.

Chapitre 7

Evaluation de JAdapt

Ce chapitre a un double objectif : montrer comment utiliser le prototype construit autour de notre modèle et présenter une évaluation de cette implémentation à travers des exemples d'utilisation. Nous rappelons que ce modèle est construit dans le but d'améliorer la réutilisation de préoccupations dans le paradigme objet. Le mot préoccupation est employé ici dans le sens donné par la séparation des préoccupations [LH 95].

Nous avons présenté : *i)* dans le chapitre 4 le cahier des charges de notre modèle, *ii)* dans le chapitre 5 le modèle proprement dit et, *iii)* dans le chapitre 6 une possible implémentation de celui-ci. Nous allons maintenant évaluer l'adéquation entre les possibilités théoriques du modèle et les possibilités offertes par l'implémentation qui est proposée.

Il est d'abord utile de rappeler que les différentes approches dédiées à la séparation des préoccupations qui ont été étudiées dans l'état de l'art sont toutes réalisées pour les langages à classes dans un environnement compilé, fortement typé. Tout naturellement, l'implémentation du modèle et l'évaluation qui est proposée dans ce chapitre se placent dans ce contexte.

L'état de l'art a permis de sélectionner un ensemble d'approches : la programmation orientée objets, la programmation générique, la métaprogrammation, la programmation par aspects, et la programmation par sujets. Elles ont toutes été étudiées en détail dans le chapitre 3.

L'étude proposée par le chapitre 3 aboutit à deux constatations par rapport à la problématique de la réutilisation : *i)* bien que l'approche objet possède deux mécanismes qui sont particulièrement intéressants pour la réutilisation à savoir l'héritage et le polymorphisme, elle possède cependant de nombreuses limitations, *ii)* les meilleurs outils trouvés dans l'état de l'art se comportent comme des surcouches de l'approche à objets et n'arrivent que partiellement à combler ces limitations. Cette double constatation a été obtenue grâce à un comparatif des propriétés de réutilisation des différentes approches. Celui-ci s'est appuyé sur trois exemples d'application : les préoccupations fonctionnelles et non fonctionnelles et les patrons de conception.

Il est donc important de confronter notre approche et son implémentation aux mêmes exemples que ceux utilisés dans le chapitre 3. Le résultat que nous en attendons est double : évaluer notre approche et la comparer aux approches étudiées dans le chapitre 3.

Ce chapitre est découpé en trois parties. La première montre comment installer et utiliser notre plug-in dans la plate-forme Eclipse [ECL 04]. La deuxième confronte notre implémentation aux exemples et aux approches décrits le chapitre 3. La troisième dresse une synthèse des résultats et propose un bilan.

7.1 INSTALLATION ET UTILISATION DE NOTRE PLUG-IN POUR ECLIPSE

Nous expliquons ici comment installer et utiliser le plug-in JAdapt dans le cadre de la plate-forme Eclipse.

7.1.1 Installation d'Eclipse

Eclipse v2.1.1 [ECL 04] (pour le téléchargement voir http://www.eclipse.org) nécessite une machine virtuelle Java compatible avec la version 1.4 de Java (voir http://java.sun.com).

Nous n'évoquerons ici, ni l'installation détaillée d'Eclipse ou d'une machine virtuelle Java, ni l'utilisation d'Eclipse pour programmer en Java. Toutes ces informations sont disponibles aux deux adresses Internet mentionées ci-dessus.

7.1.2 Installation du plug-in JAdapt

Le plug-in est disponible sous la forme d'une archive (JAdapt.zip) sur le site du projet OCL (Objets et Composants Logiciels) au laboratoire I3S (http://www.i3s.unice.fr). Cette archive doit être décompactée dans le sous répertoire `plugins` du répertoire d'installation de la plateforme Eclipse.

Une fois cette opération effectuée, il faut, le cas échéant, redémarrer la plate-forme Eclipse.

7.1.3 Utilisation du plug-in JAdapt

Pour être activé, le plug-in a besoin qu'un projet *Java* soit ouvert dans la plate-forme Eclipse (voir figure 18, projet `Exemple_PatronDeConception`).

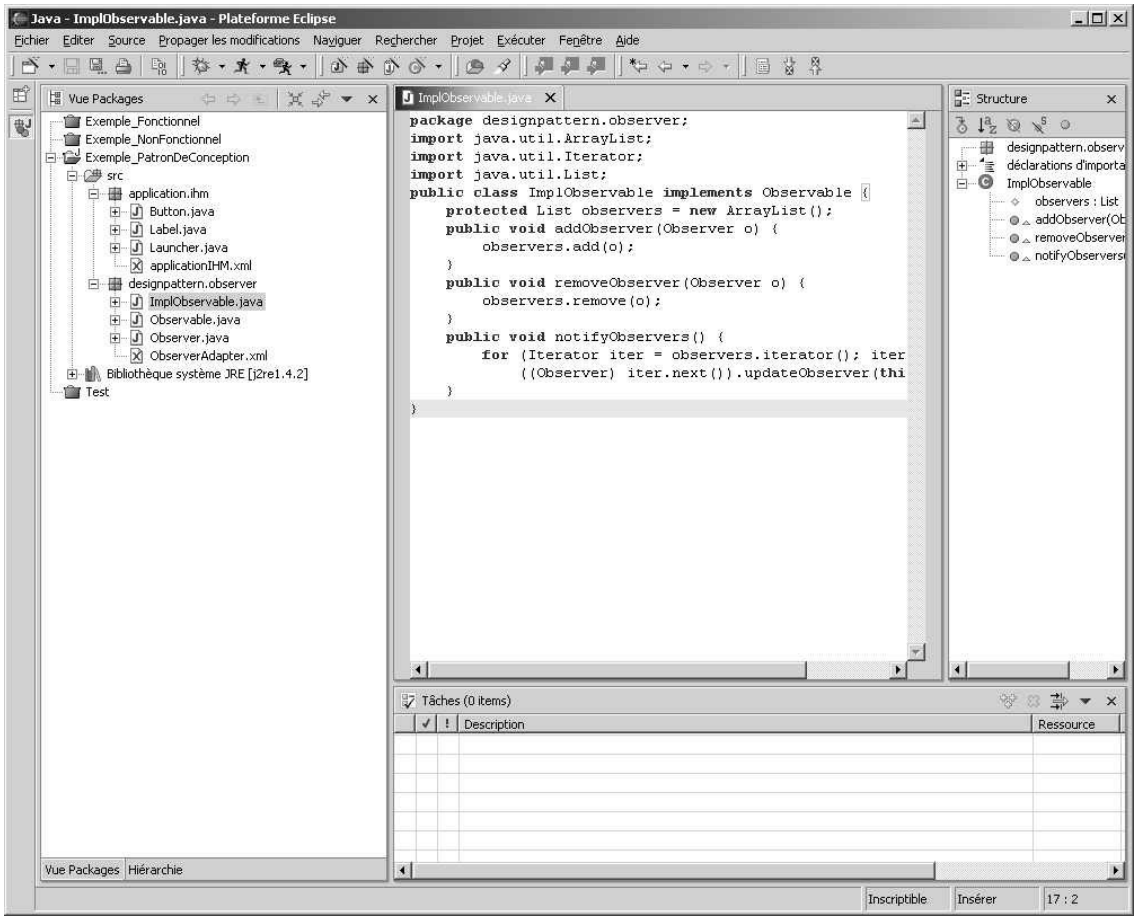


Figure 18. *Plate-forme Eclipse avec un projet Java ouvert*

Une fois le projet Java ouvert, l'utilisateur doit, pour ajouter la nature JAdapt au projet, cliquer avec le bouton droit de la souris sur le projet à convertir dans la « vue package » d'Eclipse et choisir dans le sous-menu contextuel l'option « Toggle Jadapt Project Nature » (voir figure 19).

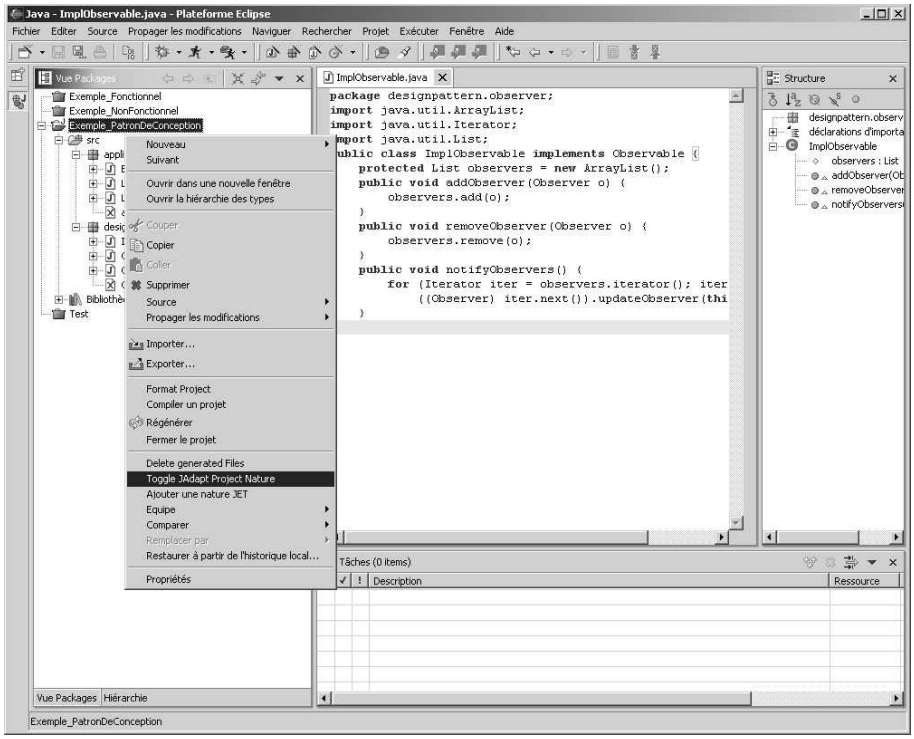


Figure 19. *Menu contextuel des projets qui permet d'ajouter JAdapt à un projet Java*

Notons que le menu contextuel de la figure 19 contient les deux seules modifications visibles de l'interface d'Eclipse qui ont été opérées par JAdapt. Ces modifications correspondent à deux nouvelles options applicables aux projets :

– `Toggle Jadapt Project Nature` : permet d'ajouter ou d'enlever le support de JAdapt à un projet Java existant.

– `Delete generated Files` : permet de supprimer les fichiers générés par JAdapt pour montrer le résultats de la composition[1].

Une fois qu'un projet est équipé pour supporter JAdapt, l'utilisateur peut définir des adaptateurs qui sont à l'heure actuelle spécifiés dans des fichiers XML. Pour cela, il suffit à partir du menu contextuel de la « vue package », de demander la création d'un nouveau fichier sans oublier d'ajouter l'extension « `.xml` » à la fin du nom du fichiers créé. Pour connaître la syntaxe de l'entête des fichiers XML correspondant à des adaptateurs on peut se référer aux différents exemples qui sont proposés dans les sections suivantes.

[1] Notons que ces fichiers ne sont pas nécessaires à l'exécution du projet. Car les fichiers adaptés sont de toute façon présents en mémoire centrale pour être utilisés par le compilateur natif Java.

Une des limitations de l'implémentation actuelle du plug-in est l'absence du support de la compilation incrémentale[1] car elle complexifie l'implémentation de la phase de composition des préoccupations et nous voulions avant tout avoir rapidement un prototype pour confronter nos idées. Ainsi il est nécessaire, à chaque modification du source ou d'un adaptateur, de « recompiler » l'ensemble du projet. Pour cela, il faut sélectionner le projet à recompiler dans la « vue package », ouvrir le menu « Projet » et choisir l'option « Recompiler » (voir figure 20).

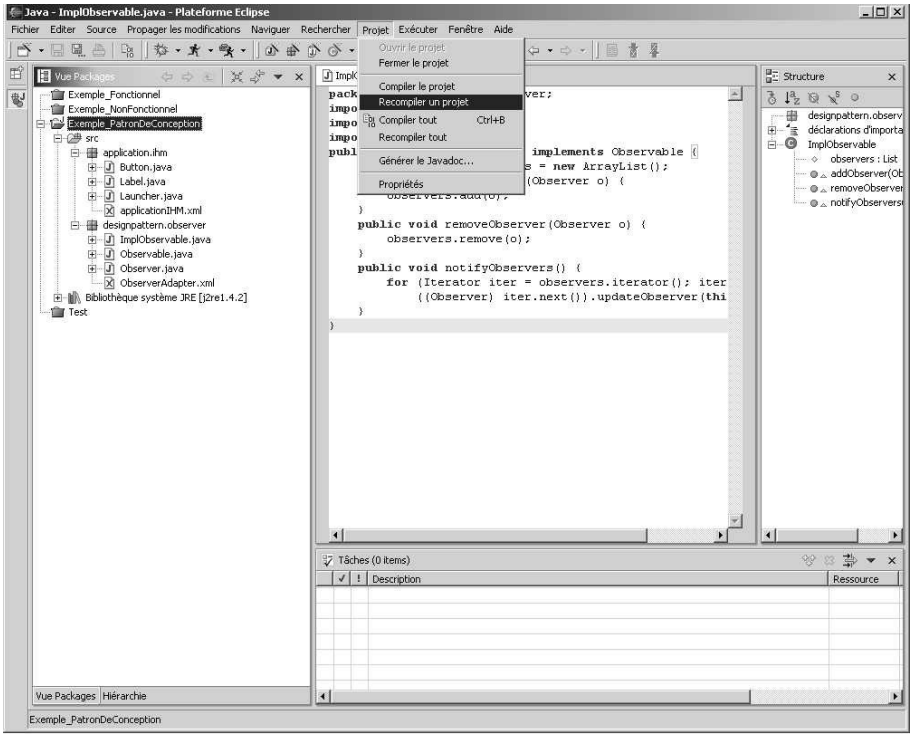


Figure 20. *Recompilation d'un projet.*

La recompilation d'un projet déclenche l'exécution de notre compilateur qui agit comme un préprocesseur du compilateur natif de la plate-forme. Cela a pour effet de compiler les fichiers après qu'ils aient été modifiés sous le contrôle des adaptateurs et des algorithmes d'adaptation (voir chapitre 5). Par ailleurs un ensemble de fichiers est généré pour chaque fichier adapté[2]. Ces fichiers contiennent les modifications réalisées sur le code source ; ils permettent à l'utilisateur de prendre connaissance des modifications effectuées.

Une application peut être exécutée soit par une ligne de commande, soit par la plate-forme Eclipse au moyen du menu « Exécuter ».

7.2 EVALUATION DE JADAPT

Nous évaluons ici les capacités de réutilisation de notre approche sur trois exemples que nous avons déjà utilisés dans le chapitre 3 pour évaluer la programmation objet, la programmation générique, la métaprogrammation, la programmation par aspects, et la programmation par sujets.

[1] Pourtant la plateforme d'Eclipse supporte la compilation incrémentale.
[2] Leur nom est de la forme `xxxx-Jadapt-Altered.Java`

Comme cela a déjà été évoqué dans le chapitre 3 notre évaluation des capacités de réutilisations relativement à la séparation des préoccupations, repose sur plusieurs critères dont les plus importants sont : *i)* la capacité de séparer des préoccupations, c'est à dire de pouvoir les exprimer de façon à ce qu'elles soient découplées du reste de l'application, *ii)* la possibilité, pour chacune des préoccupations qui a été décrite indépendamment, d'implémenter son protocole de composition ; celui-ci a pour objectif de faciliter l'intégration de la préoccupation dans l'application et donc sa réutilisation, *iii)* la faculté de pouvoir spécialiser et adapter le protocole de composition de manière à ce que toute préoccupation puisse être composée avec le reste de l'application en adéquation avec les besoins spécifiques liés à l'utilisation.

Nous défendons l'idée que le respect de ces critères de réutilisation des préoccupations rend nécessaire l'extension des langages à objets, que ces derniers possèdent ou non un niveau méta ou le support de la généricité. Nous entendons montrer ci-dessous que cette extension des langages à objet doit idéalement posséder les propriétés mises en œuvre dans notre modèle (voir chapitre 5). Pour cela nous confrontons notre implémentation du modèle à trois types de préoccupation : les préoccupations non fonctionnelles et fonctionnelles et les patrons de conception. Pour chacun de ces types de préoccupation nous montrerons les apports et les limitations de JAdapt, et nous comparerons les résultats relatifs à la capacité de réutilisation à ceux qui ont été obtenus par les approches étudiées dans le chapitre 3.

7.2.1 Préoccupations non fonctionnelles

Nous avons déjà défini dans la partie 3.1 le terme de *préoccupation fonctionnelles* d'une application. Ce sont les services qu'elle fournit (algorithme, logique métier, etc.) et ils sont généralement décrits par la spécification d'interfaces. Les *préoccupations non fonctionnelles* désignent les autres caractéristiques de l'application. Elles sont liées à la façon dont sont implémentées les propriétés fonctionnelles. On peut citer par exemple les performances, la fiabilité, la disponibilité, la qualité de service, la distribution, la persistance, les outils pour réaliser des traces, etc.

Nous étudions dans cette section le même exemple que dans la partie 3.1 : l'implémentation et la réutilisation d'une préoccupation non fonctionnelle dédiée au traçage de l'exécution d'un programme.

7.2.1.1 Implémentation et évaluation avec JAdapt

Comme cela a été mentionné ci-dessus, il faut d'abord séparer la préoccupation de l'application, c'est-à-dire éviter tout couplage que ce soit dans un sens ou dans un autre (cela constitue la première étape). Pour cela, nous isolons la préoccupation « trace » dans un paquetage Java nommée `utils.trace` qui contient une seule classe appelée `Trace`. Celle-ci possède les fonctionnalités qui permettent de tracer les appels de méthode. Le propos n'étant pas la mise en œuvre d'un mécanisme de trace complexe mais d'évaluer JAdapt, les informations produites par la trace seront envoyées directement sur la sortie standard des erreurs. Voici une implantation possible de la classe `Trace` :

```
1   Package utils.trace
2
3   public class Trace {
4
5     private static void Trace_MethodCall(String className,
6                                          String methodName,
7                                          Object[] parameters) {
8       System.err.print(className+".”+methodName+”(“);
9
10      for (int i=0; i<parameters.length ; i++) {
11         System.err.print(parameters[i]);
12         if ( i != (parameters.length-1)
13           System.err.print(“,”);
14      }
15
16      System.err.print(“)”);
17    }
18
19    public static void Trace_Beginning ( String className,
20                                         String methodName,
21                                         Object[] parameters) {
22      Trace_MethodCall(className, methodName, parameters);
23      System.err.println(“ - Beginning”);
24    }
25
26    public static void Trace_Ending( String className,
27                                     String methodName,
28                                     Object[] parameters,
29                                     Object result ) {
30      Trace_MethodCall(className, methodName, parameters);
31      System.err.print(“ = ”+result+” - ending“ );
32    }
33  }
```

La classe `Trace` possède malgré sa simplicité toutes les fonctionnalités nécessaires pour tracer des appels de méthodes : `Trace_Beginning` trace le début d'un appel de méthode et `Trace_Ending` trace la fin de l'appel de méthode. Une fois la préoccupation décrite, nous nous intéressons maintenant à la spécification du protocole de composition de la trace (c'est la deuxième étape) :

```
34  <?xml version="1.0" encoding="UTF-8" ?> <-- TraceAdaptateur.xml -->
35  <jadapt xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
36          xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
37      <package>utils.trace</package>
38      <adapter name="TraceAdapter" abstract="1">
39          <variable name="tracedMethod" type="Method"
40                    abstract="1" comment="methods to be traced">
41          </variable>
42          <variable name="tracedClass" type="Class"
43                    abstract="1" comment="classes to be traced">
44          </variable>
45          <Around name="methodCall" comment="trace's composition">
46              <pointcut>
47                  <classes>tracedClass</classes>
48                  <methods>tracedMethod</methods>
49              </pointcut>
50              <do>
51               Trace.Trace_Beginning(JointPoint_ClassName,
52                                     JointPoint_MethodName,
53                                     JointPoint_MethodParameters);
54               proceed();
55               Trace.Trace_Ending(JointPoint_ClassName,
56                                  JointPoint_MethodName,
57                                  JointPoint_MethodParameters,
58                                  result);
59             </do>
60          </Around>
61      </adapter>
62  </jadapt>
```

Cet adaptateur est écrit au format XML et respecte notre Schema-XML (voir ligne 2), mais pour des raisons de simplicité nous allons expliquer sont fonctionnement à partir de la version équivalente en JAdapt :

```
63 package utils.trace;
64 abstract adapter TraceAdapter  {
65    /**methods to be traced*/
66    Method tracedMethod;
67    /**classes to be traced*/
68    Class tracedClass;
69    /**trace's composition*/
70    around methodCall ( *.tracedClass.tracedMethod(...) )  do {
71       Trace.Trace_Beginning(JointPoint_ClassName,
72                             JointPoint_MethodName,
73                             JointPoint_MethodParameters);
74
75       proceed();
76
77       Trace.Trace_Ending(JointPoint_ClassName,
78                          JointPoint_MethodName,
79                          JointPoint_MethodParameters,result);
80    }
81 }
```

Les lignes 63 à 81 sont générées par nos soins à partir de la description XML (lignes 34 à 62) de l'adaptateur abstrait `TraceAdapter` qui par convention est défini dans le fichier `TraceAdapter.xml` lui-même contenu dans paquetage `utils.trace` (ligne 37 et 63). Ce choix de localisation permet de réunir la préoccupation et son protocole de composition.

La trace est une préoccupation non fonctionnelle qui s'exécute lors de l'appel d'une méthode et garde un historique (chronologique) de ses appels. Le protocole de composition de la trace doit offrir une composition abstraite qui permet de lier la trace (plus précisément les méthodes `Trace.Trace_Beginning` et `Trace.Trace_Ending`), à ses futurs clients. L'adaptateur `TraceAdapter` réifie le protocole de composition dans notre modèle par les éléments suivants :

– La trace s'applique à des classes. Nous avons déclaré une variable de type classe nommée `tracedClass` (voir les lignes 42 à 44 pour la description XML ou la ligne 68 pour son équivalent). A ce stade les classes à tracer ne sont pas précisées. Ainsi, en JAdapt, la variable de type Class n'est pas initialisée et en XML elle est déclarée abstraite. La variable `tracedClass` devra obligatoirement recevoir une valeur par le futur utilisateur de la préoccupation.

– La nécessité de posséder un mécanisme de trace avec un grain assez fin suggère de pouvoir préciser l'ensemble des méthodes sur lesquelles s'applique la trace. Ainsi, nous avons déclaré une variable `tracedMethod` (voir ligne 39 à 41 pour la description XML ou la ligne 66 pour son équivalent en JAdapt) toujours sans préciser de valeur[1].

– L'utilisation d'une adaptation de type *interception* autour de l'appel de méthode (utilisation d'**`around`**) qui est nommée `methodCall` (ligne 45 à 60 pour la description XML ou lignes 70 à 80 pour son équivalent en JAdapt) permet de définir la composition des clients de la trace avec la préoccupation. La description de cette adaptation nécessite deux précisions complémentaires :

  – la spécification d'un point de jointure (lignes 46 à 49 pour la description XML ou ligne 70 pour son équivalent en JAdapt) qui indique à l'adaptateur où effectuer les modifications : ici, la description du point de jointure utilise les deux variables définies précédemment.

  – La description d'un bout de code[2] (lignes 50 à 59 pour la description XML ou lignes 71 à 79 pour son équivalent en JAdapt) qui est exécuté quand le programme at-

[1] Pour les mêmes raisons que la variable `tracedClass`.

[2] Dans notre implémentation ce bout de code est décrit en Java que ce soit dans le fichier XML ou dans l'adaptateur généré.

teint le point de jointure. Il est à noter que le point de jointure, par l'intermédiaire des pseudos-variables `JointPointxxx_`et `result`, fournit à la classe `Trace` les informations contextuelles qui lui sont nécessaires. Le pseudo appel de méthode `proceed()` représente l'appel du code initial de la méthode interceptée.

Les trois éléments du protocole de composition (abstrait) sont correctement réifiés par JAdapt : l'ensemble des classes et des méthodes à tracer et le code nécessaire à l'exécution de la trace.

Comme dans la section 3.1 nous allons appliquer la Trace à une classe Image :

```
82  package application;
83
84  public class Image {
85      private int[][] pixels;
86
87      public Image(int width,int height) {
88        pixels=new int[width][height];
89      }
90
91      public static void main (String args []) {
92        (new Image(10,10)).rotation(90);
93      }
94
95      public void rotation(int angle) {
96        //Allocate another array of pixel
97        for (int x=0; x<pixels.length;x++)
98          for (int y=0; y<pixels[x].length;y++) {
99            //do some stuff
100       }
101       //Replace the original array of pixels by the new one
102     }
103 }
```

La classe `Image` fait partie du paquetage `application`, tout comme l'adaptateur concret `TraceMonImage` qui est décrit ci-dessous sous forme XML. L'objectif de ce dernier est de tracer la méthode `rotation` de la classe `Image` :

```
104 <?xml version="1.0" encoding="UTF-8" ?>
105 <jadapt xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
106         xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
107     <package>application</package>
108     <import>utils.trace.*</import>
109     <adapter name="TraceMonImage" extends="TraceAdapter">
110         <variable name="tracedMethod" type="Method">rotation
111         </variable>
112         <variable name="tracedClass" type="Class">Image
113         </variable>
114     </adapter>
115 </jadapt>
```

Cet adaptateur peut aussi s'exprimer en JAdapt :

```
116 package application;
117
118 import utils.trace.*;
119
120 adapter TraceMonImage extends TraceAdapter  {
121     Method tracedMethod = rotation;
122     Class tracedClass = Image;
123 }
```

L'adaptateur `TraceMonImage` hérite (voir ligne 109 pour la description XML ou 120 pour son équivalent en JAdapt) de l'adaptateur `TraceAdapter` et le spécialise pour le cas d'utilisation mentionné. On a vu plus haut que l'adaptateur abstrait dont il hérite possède deux variables sans

valeur. Pour devenir un adaptateur concret `TraceMonImage` doit donner une valeur à chacune des variables ; JAdapt s'assure que c'est le cas lors de la phase de vérification du typage des adaptateurs. La compilation de l'application est interrompue si l'utilisateur n'a pas correctement spécialisé tous les éléments abstraits de son protocole de composition et il est alors averti par un message approprié pour chacune de ses erreurs directement par la plate-forme Eclipse.

De plus, ces deux variables comme tous les points de jointure supportent des expressions régulières (qui sont exprimées dans la syntaxe de l'API `java.util.regex`). Dans l'adaptateur `TraceMonImage` nous aurions pu donner par exemple `rotation.*` comme valeur à la variable `tracedMethod` pour tracer toutes les méthodes donc le nom commence par le mot rotation.

La composition des préoccupations de notre implémentation modifie la classe `Image` (voir lignes 82 à 103) et génère automatiquement le code présenté ci-dessous :

```
124 package application;
125 import utils.trace.*;
126 public class Image {
127     private int[][] pixels;
128     public Image (int width, int height) {
129       pixels = new int[width][height];
130     }
131     public static void main(String args[]) {
132       (new Image (10, 10)).rotation(90);
133     }
134     public void rotation(int angle) {
135       String JointPoint_ClassName = "Image";
136       String JointPoint_MethodName = "rotation";
137       Object[] JointPoint_MethodParameters = new Object[] { new Integer(angle)};
138       Trace.Trace_Beginning(JointPoint_ClassName,
139                             JointPoint_MethodName,
140                             JointPoint_MethodParameters);
141       rotation_Jadapt_0(angle);
142       Object result = null;
143       Trace.Trace_Ending(JointPoint_ClassName,
144                          JointPoint_MethodName,
145                          JointPoint_MethodParameters,
146                          result);
147     }
148     private void rotation_Jadapt_0(int angle) {
149       //Allocate another array of pixel
150       for (int x=0; x<pixels.length;x++)
151         for (int y=0; y<pixels[x].length;y++) {
152           //do some stuff
153       }
154       //Replace the original array of pixels by the new one
155     }
156 }
```

Il faut d'abord préciser que le code source ci-dessus remplace le code source orignal de la classe `Image` mais seulement pendant la phase de compilation. L'adaptateur `TraceMonImage` produit une composition *in-situ* car le fichier XML correspondant à l'adaptateur a été placé dans le même paquetage que celui où se trouve l'application (voir ligne 116).

La première adaptation effectuée par `TraceMonImage` consiste à inclure toutes les importations définies par l'adaptateur concret et ceux dont il hérite (dans notre cas il y a donc aussi `TraceAdapter`). Ceci a pour conséquence d'importer le paquetage `utils.trace` (voir ligne 124), permettant ainsi à la classe `Image` de faire référence à la classe `Trace`.

La méthode `rotation` de la classe `Image` est ensuite interceptée par l'adaptation de type **`around`** (voir lignes 70 à 80). Cette adaptation renomme la méthode d'origine (voir lignes 148 à 155) et restreint sa visibilité, le nouveau nom est choisi de façon à éviter tout conflit de nom. Une nouvelle méthode `rotation` est générée ; elle possède la même signature que la méthode d'origine mais son code source correspond au code spécifié par les lignes 71 à 79 de l'adaptateur

`TraceAdapter`. Enfin, des informations contextuelles[1] sur le point de jointure sont ajoutées en fonction de l'utilisation d'une des trois pseudo-variables (voir lignes 135 à 137) par le code d'interception. Dans le cas présent, le code d'interception (voir lignes 70 à 80) utilise les trois pseudo-variables.

L'exemple présenté ci-dessus illustre la possibilité d'une part de séparer la préoccupation « trace » de ses applications et d'autre part de la réutiliser dans une application sans modifier cette dernière. Ceci est possible parce que *i)* l'encapsulation complète du protocole de composition abstrait est réalisée dans un adaptateur abstrait et *ii)* les informations qui sont spécifiques à l'application sont placées dans un adaptateur concret qui hérite du précédent et complète le protocole de composition pour le mettre en adéquation avec l'application.

7.2.1.2 Bilan relatif aux préoccupations non fonctionnelles

L'étude développée dans la section 3.1 a permis de répondre à trois questions : *i)* ces préoccupations sont-elles séparables ? *ii)* si c'est le cas, comment les séparer ? *iii)* est-il facile de réutiliser les préoccupations non fonctionnelles une fois qu'elles ont été séparées ?

La section 3.1.1 a confirmé que le paradigme de l'objet (avec ou sans support de la généricité) ne permet pas de séparer les préoccupations non fonctionnelles des classes concernées. La section 3.1.2 a montré que la métaprogrammation [KRB 91] permet de séparer les préoccupations non fonctionnelles mais en introduisant une plus grande complexité ce qui a pour conséquence de rendre problématique la réutilisation des préoccupations non fonctionnelles. La section 3.2.3 a montré que la programmation par aspects [KLM 97] permet de séparer les préoccupations non fonctionnelles, et de les réutiliser de manière aisée. Enfin, la section 3.1.4 a montré que la programmation par sujets [HO 93] permet de séparer les préoccupations non fonctionnelles mais leur réutilisation reste peu pratique.

La conclusion émise dans le chapitre 3, est que parmi les approches mentionnées, seule la programmation par aspects permet de séparer et de réutiliser facilement des préoccupations non fonctionnelles. Comment se positionne JAdapt par rapport à cette approche ? Une première réponse est que les résultats produits par la programmation par aspects (AspectJ) sont identiques à ceux de JAdapt :

– La préoccupation de la trace a été correctement séparée du reste de l'application, en effet :
  – La classe trace est isolée dans le paquetage `utils.trace` sans qu'il subsiste un couplage qui ne soit pas désiré.
  – Le protocole de composition de la trace a été entièrement rendu abstrait et ses points d'adaptation ont été tous correctement explicités.
– La préoccupation a été composée avec le reste de l'application sans induire de couplage inutile. La classe `Image` a été adaptée automatiquement en complétant le protocole de composition par un adaptateur (ou un aspect) concret.

AspectJ et JAdapt possèdent les capacités nécessaires pour séparer et composer des préoccupations non fonctionnelles et facilitent leur réutilisation.

7.2.2 Préoccupations fonctionnelles

Les préoccupations fonctionnelles correspondent aux fonctionnalités fournies par une application, autrement dit à sa logique métier, c'est-à-dire la représentation et le traitement de l'information directement en relation avec l'objectif principal de l'application. Bien que les préoc-

[1] Qui sont utilisable par les préoccupations, comme dans le cas présent par la trace pour savoir qu'elle est la méthode tracée.

cupations fonctionnelles semblent plus dépendantes de l'application, il est possible d'envisager leur partage par un ensemble d'applications.

JAdapt va être utilisé afin de séparer et de composer ensemble trois préoccupations fonctionnelles déjà étudiées dans la section 3.2 : la structure de l'arbre des expressions arithmétiques (encapsulée par le paquetage `expr.syntaxe`), l'évaluation des expressions arithmétiques (encapsulée par le paquetage `expr.eval`) et l'affichage de structures arborescentes (paquetage `pattern.affichageast`).

7.2.2.1 Implémentation et évaluation avec JAdapt

Le chapitre 3 a montré que pour pouvoir être pleinement réutilisable, ce type de préoccupation doit être composé de manière *ex-situ*. Cela signifie que le résultat de leur composition ne doit pas modifier les préoccupations et donc que ces modifications doivent être placées ailleurs, par exemple dans un nouveau paquetage.

Parmi les trois préoccupations citées, seul l'affichage de structures arborescentes peut être doté d'un protocole de composition car c'est la seule préoccupation qui n'est pas directement utilisable. Elle nécessite en effet d'être composée à d'autres préoccupations d'une application car elle représente une fonctionnalité à ajouter qui n'est pas directement utilisable telle qu'elle.

L'affichage de structures arborescentes nécessite donc un protocole de composition qui contient les adaptations suivantes :

– Identifier les classes jouant un des rôles spécifiés par la préoccupation : *node*, *leaf* ou *composite*, chaque classe jouant un rôle doit être adaptée.

– Abstraire l'implémentation des trois méthodes suivantes : `ASTNode.displayType()`, `ASTLeaf.displayValue()`, et `ASTComposite.children()`.

L'implémentation de ce protocole de composition sous la forme d'un fichier XML pour JAdapt est présentée ci-dessous :

```
1   <?xml version="1.0"  encoding="UTF-8" ?>
2   <jadapt xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
4       <package>framework.affichageast</package>
5       <adapter name="ProtocoleAffichageAST" abstract="1">
6           <variable name="Node" type="Class" abstract="1"
7                     comment="class being used as a Noce"/>
8           <variable name="Leaf" type="Class" abstract="1"
9                     comment="class being used as a Leaf"/>
10          <variable name="Composite" type="Class" abstract="1"
11                    comment="class being used as a Composite"/>
12          <fusion name="becomeNode"
13              comment="Modifying class in order to become a Node">
14              <source>ASTNode</source>
15              <target>Node</target>
16          </fusion>
17          <fusion name="becomeLeaf"
18              comment="Modifying class in order to become a Leaf">
19              <source>ASTLeaf</source>
20              <target>Leaf</target>
21          </fusion>
22          <fusion name="becomeComposite"
23            comment="Modifying class in order to become a Composite">
24              <source>ASTComposite</source>
25              <target>Composite</target>
26          </fusion>
27         <introduce name="nodeDisplay" abstract="1"
28           comment="Introduce displayType() method in the Node class">
29            <pointcut>
30              <classes>Node</classes>
31            </pointcut>
32            <method>
```

```
33              <name>public void displayType</name>
34              <parameters></parameters>
35            </method>
36         </introduce>
37         <introduce name="leafDisplayValue" abstract="1"
38      comment="Introduce displayValue() method in the Leaf class">
39           <pointcut>
40              <classes>Leaf</classes>
41           </pointcut>
42           <method>
43            <name>public void displayValue</name>
44              <parameters></parameters>
45           </method>
46         </introduce>
47         <introduce name="compositeChildren" abstract="1"
48     comment="Introduce children() method in the Children class">
49           <pointcut>
50             <classes>Composite</classes>
51           </pointcut>
52           <method>
53             <name>public ASTNode[] children</name>
54             <parameters></parameters>
55           </method>
56         </introduce>
57      </adapter>
58  </jadapt>
```

La description proposée ci-dessous représente le même adaptateur mais décrit avec le langage de plus haut niveau de JAdapt (notons que ce langage est aussi dénommé JAdapt) :

```
59  package framework.affichageast;
60  abstract adapter ProtocoleAffichageAST  {
61    /**class being used as a Node*/
62    Class Node;
63
64    /**class being used as a Leaf*/
65    Class Leaf;
66
67    /**class being used as a Composite*/
68    Class Composite;
69
70    /**Modifying class in order to become a Node*/
71    fusion(becomeNode) ASTNode in Node;
72
73    /**Modifying class in order to become a Leaf*/
74    fusion(becomeLeaf) ASTLeaf in Leaf;
75
76    /**Modifying class in order to become a Composite*/
77    fusion(becomeComposite) ASTComposite in Composite;
78
79    /**Introduce the display() method in the Node class*/
80    introduce(nodeDisplay) public void displayType() in Node;
81
82    /**Introduce the displayValue() method in the Leaf class*/
83    introduce(leafDisplayValue) public void displayValue() in Leaf;
84
85    /**Introduce the children() method in the Children class*/
86    introduce(compositeChildren) public ASTLeaf[] children()
87       in Composite;
88  }
```

On retrouve dans l'adaptateur décrit ci-dessus tous les éléments nécessaires au protocole de composition de la préoccupation :

– les trois variables de type `Class` représentent les classes clientes et jouent chacune l'un des trois rôles nécessaires à la préoccupation (voir lignes 6 à 11 pour la description XML ou 61 à 68 pour son équivalent en JAdapt) ;

– la fusion des classes de la préoccupations dans les classes clientes (voir lignes 12 à 26 la description XML ou 70 à 77 pour son équivalent en JAdapt) ;

– la nécessité pour les classes clientes d'implémenter trois méthodes (voir lignes 27 à 57 pour la description XML ou 89 à 87 pour son équivalent en JAdapt).

JAdapt permet donc d'encapsuler complètement le protocole de composition de la préoccupation et de réaliser l'affichage des structures arborescentes. Ce protocole de composition va pouvoir être réutilisé par héritage dans un nouvel adaptateur qui va composer directement les trois préoccupations.

```
89  <?xml version="1.0"  encoding="UTF-8" ?>
90  <jadapt xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
91          xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
92      <package>application.ast.expressionarithmetique</package>
93      <import>framework.affichageast.*</import>
94      <import>expr.syntaxe.*</import>
95      <import>expr.eval.*</import>
96      <adapter name="ExpressionArithmetiqueAffichable"
97               extends="ProtocoleAffichageAST">
98      <!--Merging expr.syntaxe and expr.eval -->
99        <fusion name="mergeExpr">
100         <source>expr.eval.expr</source>
101         <target>expr.syntaxe.expr</target>
102       </fusion>
103       <fusion name="mergeNum">
104         <source>expr.eval.Num</source>
105         <target>expr.syntaxe.Num</target>
106       </fusion>
107       <introduce name="adaptNum">
108         <pointcut>
109           <classes>application.ast.expressionarithmetique.Num
110           </classes>
111         </pointcut>
112         <method>
113           <name>public integer value()</name>
114           <parameters></parameters>
115           <body> return new Integer(getValue());</body>
116         </method>
117       </introduce>
118       <fusion name="mergeOpBinaire">
119         <source>expr.eval.OpBinaire</source>
120         <target>expr.syntaxe.OpBinaire</target>
121       </fusion>
122       <fusion name="mergeOpUnaire">
123         <source>expr.eval.OpUnaire</source>
124         <target>expr.syntaxe.OpUnaire</target>
125       </fusion>
126       <!--Merging application.ast.expressionarithmetique and
127         framework.affichageAST -->
128       <variable name="Node" type="Class">
129           application.ast.expressionarithmetique.Expr</variable>
130       <variable name="Leaf" type="Class">
131           application.ast.expressionarithmetique.Num</variable>
132       <variable name="Composite" type="Class">
133          application.ast.expressionarithmetique.OpBinaire AND
134          application.ast.expressionarithmetique.OpUnaire</variable>
135       <introduce name="nodeDisplay">
136         <pointcut>
137           <classes>Node</classes>
138         </pointcut>
139         <method>
140           <name>public void displayType</name>
141           <parameters></parameters>
142           <body>Sysem.out.println(getType());</body>
143         </method>
144       </introduce>
145       <introduce name="leafDisplayValue">
146         <pointcut>
147           <classes>Leaf</classes>
148         </pointcut>
149         <method>
150           <name>public void displayValue</name>
151           <parameters></parameters>
152           <body>System.out.println(getValue());</body>
153         </method>
154       </introduce>
155       <introduce name="compositeChildren">
156         <pointcut>
```

```
157           <classes>Composite</classes>
158         </pointcut>
159         <method>
160           <name>public ASTNode[] children</name>
161           <parameters></parameters>
162           <body>if ( this instanceof OpBinaire)
163            return new ASTNode[] {
164              ((OpBinaire)this).getLeft(),
165              ((OpBinaire)this).getRight() };
166            else return new ASTNode[] { ((OpUnaire)this).getOp() };
167           </body>
168         </method>
169      </introduce>
170     </adapter>
171 </jadapt>
```

Comme nous l'avons fait pour l'adaptateur abstrait, la description proposée ci-dessous est présentée ensuite dans le langage de plus haut niveau de JAdapt :

```
172 package application.expr;
173 import framework.affichageast.*;
174 import expr.syntaxe.*;
175 import expr.eval.*;
176 adapter ExpressionArithmetiqueAffichable
177   extends ProtocoleAffichageAST  {
178   fusion(mergeExpr) expr.eval.expr in expr.syntaxe.expr;
179   fusion(mergeNum) expr.eval.Num in expr.syntaxe.Num;
180   introduce(adaptNum) public Integer value() {
181    return new Integer(getValue()); }  in application.expr.Num;
182
183   fusion(mergeOpBinaire) expr.eval.OpBinaire
184     in expr.syntaxe.OpBinaire;
185   fusion(mergeOpUnaire) expr.eval.OpUnaire
186     in expr.syntaxe.OpUnaire;
187
188   Class Node = application.expr.Expr;
189   Class Leaf = application.expr.Num;
190   Class Composite =  application.expr.OpBinaire AND
191                      application.expr.OpUnaire;
192
193   introduce(nodeDisplay) public void displayType() {
194     Sysem.out.println(getType()); }  in  Node;
195
196   introduce(leafDisplayValue) public void displayValue() {
197     System.out.println(getValue()); }  in  Leaf;
198
199   introduce(compositeChildren) public ASTNode[] children() {
200   if ( this instanceof OpBinaire)
201            return new ASTNode[] { ((OpBinaire)this).getLeft(),
202                                   ((OpBinaire)this).getRight() };
203            else return new ASTNode[] { ((OpUnaire)this).getOp() };
204   }  in Composite ;
205 }
```

L'adaptateur `ExpressionArithmetiqueAffichable` (voir lignes 172 à 205) permet de composer ensemble nos trois préoccupations dans un nouveau paquetage nommée `application.expr` (voir ligne 172). Il est composé de deux parties : la première (voir lignes 178 à 186) permet de composer ensemble `expr.syntaxe` et `expr.eval`, la seconde (voir lignes 188 à 204) permet de composer le resultat de la partie précédente avec `pattern.affichageAST`.

La première partie ne contient que des adpatations de type **`fusion`** qui permettent de fusionner ensemble les classes relatives à la même entité des paquetages `expr.syntaxe` et `expr.eval`. On remarque qu'en ce qui concerne l'adaptation nommée `adaptNum` (voit lignes 180 à 181) il est nécessaire de fournir une implémentation à la méthode abstraite décrite dans la classe `expr.eval.Num`.

La deuxième partie consiste à concrétiser l'adapteur abstrait `ProtocoleAffichageAST` pour integrer dans le résultat de la première partie (qui est placé dans le paquetage `application.expr` comme le montre la ligne 172) la préoccupation d'affichage des structures arborescentes. Comme nous l'avons vu précédemment le protocole de composition nécessite : *i)* de définir les classes sur lesquelles porte la préoccupation (ceci correspond aux lignes 188 à 191), *ii)* d'implémenter trois méthodes dans les classes de *i)* ; ceci est fait à travers les lignes 193 à 204.

Le résultat de la composition générée par l'adaptateur `ExpressionArithmetique-Affichable` sur la classe `Expr` est présenté ensuite. Nous ne détaillerons pas les autres classes produites par la composition car la classe suivante est suffisamment représentative de l'adaptation effectuée.

```
206 package application.expr;
207 public abstract class Expr {
208   protected String type;
209   public Expr(String type) {
210     this.type=type;
211   }
212   public String getType() {
213     return type;
214   }
215   public abstract int eval();
216   public void Display() {
217     displayType();
218   }
219   public void displayType() {
220     System.out.println(getType());
221   }
222 }
```

La classe `Expr` du paquetage `application.expr` est le résultat de la fusion de trois classes : `expr.syntaxe.exp` (voir ligne 178), `expr.eval.expr` (voir ligne 178), et `pattern.affichageAST.ASTNode` (voir lignes 188 et 71, car une partie est décrite par l'adaptateur abstrait).

La première partie (voir lignes 208 à 214) correspond au contenu de la classe `expr.syntaxe.exp`. La ligne 215 provient de la classe `expr.eval.expr`. Les lignes 216 à 218 appartiennent à la classe `pattern.affichageAST.ASTNode`. Enfin les lignes 219 à 221 sont le résultat d'une adaptation (voir lignes 193 à 194) qui est nécessaire pour intégrer correctement la préoccupation d'affichage relative aux structures arborescentes.

Le résultat est similaire pour les autres classes (qui non pas été mentionnées). Les seuls liens d'héritage qui sont conservés sont ceux du paquetage `expr.syntaxe.exp` car les classes de ce paquetage apparaissent toujours au début des instructions d'adaptation. En effet, le résultat d'une fusion est toujours placé dans la première classe de l'ensemble ordonné des classes à fusionner (voir section 5.3.4.3). Le résultat final correspond donc au paquetage `expr.syntaxe.exp` dans lequel on aurait fusionné les classes des deux autres paquetages.

L'adaptateur `ExpressionArithmetiqueAffichable` a permis de composer nos trois préoccupations sans les modifier car le résultat est produit dans un nouveau paquetage. Cette composition a été facilitée car il a été possible de spécialiser par héritage le protocole de composition d'une des préoccupations à composer.

7.2.2.2 Bilan relatif aux préoccupations fonctionnelles

La section 3.2 a étudié la réutilisation des préoccupations fonctionnelles pour le paradigme de l'objet, la métaprogrammation, les aspects et les sujets. Nous rappelons brièvement les résultats de ces quatre approches.

Le paradigme de l'objet n'apporte aucune solution qui permettrait d'éviter à la fois la duplication de code ou le couplage entres les préoccupations fonctionnelles. Cette absence de solution rend impossible son utilisation pour séparer des préoccupations fonctionnelles.

La métaprogrammation apporte une seule solution : utiliser sa capacité générative pour produire un nouvel ensemble de classes correspondant à la composition des préoccupations fonctionnelles. Cependant cette solution reste non satisfaisante car *i)* la composition ne peut pas être abstraite, elle dépend en effet de l'ensemble des préoccupations à composer et *ii)* la composition est complexe car il faut pleinement utiliser le niveau méta.

La programmation par aspects n'apporte rien de plus aux solutions fournies par une approche par objets car ce paradigme repose sur un mode de composition uniquement *in-situ*.

La programmation par sujets permet de séparer les préoccupations fonctionnelles et de les composer à nouveau car ce paradigme utilise un mode de composition *ex-situ*. Par contre, elle n'est pas pleinement satisfaisante car : *i)* le manque de capacité d'adaptation des opérateurs de composition oblige l'utilisateur à décrire des classes dites de transition pour composer convenablement les préoccupations, et *ii)* cette approche ne permet pas l'abstraction de la composition et il n'est donc pas possible d'encapsuler le protocole de composition de la préoccupation d'affichage relative à une structure arborescente.

En conclusion, aucune des approches étudiées dans la section 3.2 ne permet de réutiliser de manière satisfaisante les préoccupations fonctionnelles. Alors que, comme nous venons de le voir dans la section précédente, JAdapt a permis d'encapsuler le protocole de composition de la préoccupation d'affichage et de composer ensemble les trois préoccupations sans que le programmeur ait besoin (comme avec Hyper/J) de définir des classes de transition.

7.2.3 Les patrons de conception

Les patrons de conception représentent un ensemble de solutions génériques pour une famille de problèmes. Ils sont fréquemment utilisés dans le processus de construction du logiciel car ils contribuent à la réutilisation d'un savoir faire. Cependant, leur implémentation dans le paradigme objet est plutôt transversale à la hiérarchie de classes et a pour conséquence de « polluer » le code source de ces classes. Cela nous a amené à choisir dans la section 3.3 de considérer leur l'implémentation comme un exemple intéressant d'application des approches par séparation des préoccupations. Ainsi l'implémentation d'un patron de conception et ses utilisations dans différents contextes doivent être clairement séparées, pour éviter d'altérer la lisibilité du code.

L'exemple du patron de conception de « *l'observateur* » utilisé par la suite provient de la section 3.3. Une implémentation avec JAdapt va être étudiée. Comme cela a été évoqué dans cette section, ce patron de conception est suffisamment représentatif car son utilisation nécessite l'adaptation de ses clients d'un point de vue à la fois fonctionnel (en opérant sur les classes) et comportemental ou non fonctionnel (en opérant sur les méthodes).

Le patron de conception observateur « définit une interdépendance de type *1 à plusieurs*, de telle façon que, lorsqu'un objet change d'état, tous ceux qui dépendent de lui soient notifiés de ces modifications et automatiquement mis à jour » [GHJ 99]. Sa mise en œuvre est décomposée en deux parties :

– Une partie composée des trois classes (voir figure 5) qui réifie les éléments du patron. Elle constitue son protocole : une entité *observable* permet à d'autres entités, les *observateurs*, de s'abonner pour recevoir des informations sur les modifications de son état. Cette partie est indépendante des classes qui vont réutiliser le patron.

– Une partie dépendante de l'utilisation du patron formée de quatre adaptations. Trois sont fonctionnelles[1] et correspondent à l'état de l'objet observé, aux liens d'héritage qui sont à modifier, et à la méthode `miseAJour()` qu'il faut implémenter dans les objets « observateurs ». La quatrième est comportementale[2] et correspond à l'ajout dans l'objet « observable » d'appels à la méthode `notifieObservateurs()` qui prévient les observateurs des changements d'état.

Nous gardons le même exemple d'utilisation qui est celui de la section 3.3. Il s'agit d'une interface homme-machine qui encapsule une interaction entre un bouton (représenté par la classe `Button`) et un label (représenté par la classe `Label`) qui joue le rôle d'observateur. Ainsi quand on clique sur un bouton (appel à la méthode `fireActionPerformed()`) le label qui l'observe a vocation à modifier son apparence (appel de la méthode `setText()`).

7.2.3.1 Implémentation et évaluation avec JAdapt

Considérons tout d'abord, les classes `Observer`, `Observable`, et `ImplObserver` qui représentent le patron de conception, ces trois classes correspondent à la préoccupation que nous voulons séparer de l'application. Le code source de ces classes est décrit ci-après :

```
1   package designpattern.observer;
2   public interface Observable {
3       public void addObserver(Observer o);
4       public void removeObserver(Observer o);
5       public void notifyObservers();
6   }
```

```
7   package designpattern.observer;
8   public interface Observer {
9       public void updateObserver(Observable o);
10  }
```

```
11  package designpattern.observer;
12  import java.util.ArrayList;
13  import java.util.Iterator;
14  import java.util.List;
15  public class ImplObservable implements Observable {
16      protected List observers = new ArrayList();
17      public void addObserver(Observer o) {
18              observers.add(o);
19      }
20      public void removeObserver(Observer o) {
21              observers.remove(o);
22      }
23      public void notifyObservers() {
24              for (Iterator iter = observers.iterator();
25                 iter.hasNext();)
26                      ((Observer) iter.next()).
27                     updateObserver(this);
28      }
29  }
```

Les deux interfaces `Observer` et `Observable` représentent les deux rôles mis en œuvre par le patron de conception. La classe `ImplObservable` représente une implémentation générique pour l'entité observable. Maintenant, nous proposons une implémentation au format XML de l'adaptateur abstrait qui représente le protocole de composition de la préoccupation :

[1] L'adaptation fonctionnelle modifie les fonctionnalités d'une classe : ses méthodes, ses variables et son graphe d'héritage.
[2] L'adaptation comportementale modifie les fonctionnalités existantes d'une classe en modifiant ses méthodes par l'ajout de code avant, après ou autour d'elle.

```
<?xml version="1.0"  encoding="UTF-8" ?>
<jadapt xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
        xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
  <package>designpattern.observer</package>
  <adapter name="ObserverAdapter" abstract="1">
    <!-- Observable's Part -->
    <variable name="observableClass" type="Class" abstract="1"
         comment="class being used as an observable"></variable>
    <variable name="notifyingMethods" type="Method" abstract="1"
         comment="methods fireing observer's changes"></variable>
    <fusion name="becomeObservable"
          comment="Modifying class in order to become observable">
       <source>ImplObservable</source>
       <target>observableClass</target>
    </fusion>
    <After name="notifyingObserver" comment="Alter notifyingMethods
         in order to fire an abservable's state change">
      <pointcut>
        <classes>observableClass</classes>
        <methods>notifyingMethods</methods>
      </pointcut>
      <do> notifyObservers(); </do>
    </After>
    <!-- Observer's Part -->
    <variable name="observerClass" type="Class" abstract="1"
        comment="class being used as an observer"></variable>
    <implement name="becameObserver"
        comment="Modifying class in order to become observer">
      <interface>Observer</interface>
      <in>observerClass</in>
    </implement>
    <introduce name="observerUpdate" abstract="1"
          comment="Introduice the  updateObserver method in the
                   observer class">
      <pointcut>
         <classes>observerClass</classes>
      </pointcut>
      <method>
        <name>updateObserver</name>
        <parameters>Observable o</parameters>
        <body></body>
      </method>
    </introduce>
  </adapter>
</jadapt>
```

Comme on peut le constater ci-dessus, cet adaptateur est décrit en utilisant le langage XML. Nous proposons aussi une version équivalente correspondant à une description en JAdapt de cet adaptateur, qui est plus proche du monde de la programmation :

```
75 package designpattern.observer;
76 abstract adapter ObserverAdapter  {
77   /**class being used as an observable*/
78   Class observableClass;
79
80   /**methods fireing observer's changes*/
81   Method notifyingMethods;
82
83   /**Modifying class in order to become observable*/
84   fusion(becomeObservable) ImplObservable in observableClass;
85
86   /**Alter notifyingMethods in order to fire an abservable's state
87    *change*/
88   after notifyingObserver(observableClass.notifyingMethods(...))do{
89     notifyObservers();
90   }
91
92   /**class being used as an observer*/
93   Class observerClass;
94
95   /**Modifying class in order to become observer*/
96   implement(becameObserver) Observer in observerClass;
97
98   /**Introduice the updateObserver method in the observer class*/
99   introduce(observerUpdate)
100     public void updateObserver(Observable o ) in
101     ( *.observerClass.*(...) )  ;
102 }
```

Les lignes 75 à 102 sont générées automatiquement à partir de la description XML de l'adaptateur abstrait `ObserverAdapter` (lignes 30 à 74) qui par convention, est défini dans le fichier `ObserverAdapter.xml` lui-même contenu dans le répertoire où se trouve le paquetage `designpattern.observer` (lignes 33 et 75). Ceci est un moyen utilisé pour réunir la préoccupation et son protocole de composition.

L'adaptateur abstrait encapsule le protocole de composition du patron de conception observateur ; il est composé de deux parties :

– La première (voir lignes 77 à 90) encapsule la composition des entités observables et contient :

– Une variable représentant une classe (`observableClass`, voir ligne 78) permet de désigner la classe à rendre observable. Cette variable contient la classe jouant le rôle d'observable. Elle est utilisée par l'adaptation (voir ligne 84) qui introduit la classe `ImplObservable` dans la classe représentée par la variable. Cette adaptation permet ainsi de rendre ces classes observables.

– Une nouvelle variable (`notifyingMethods`, voir ligne 81) désigne les méthodes qui déclenchent une mise à jour des observateurs. Cette variable permet d'implémenter l'adaptation (voir ligne 88 à 90) indépendamment de ces méthodes. L'adaptation est de type *interception de méthode* ; elle permet d'exécuter la mise à jour à la fin de l'exécution des méthodes.

– La deuxième partie (voir lignes 92 à 101) encapsule la composition des entités qui jouent le rôle d'observateur ; elle contient :

– Une variable désigne la classe qui doit se comporter comme un observateur ; il s'agit d'`observerClass` (voir ligne 93). Cette variable permet d'implémenter, indépendamment des classes « observatrices », l'adaptation (voir ligne 96) qui introduit dans les classes de `observerClass` un lien d'implémentation vers l'interface `Observer`.

– Cette même variable est réutilisée dans la description d'une autre adaptation (voir ligne 99 à 101). Celle-ci est de type *ajout de méthode* ; comme son nom l'indique, elle permet d'ajouter la méthode qui mettra à jour les observateurs. Ceci permet aux classes qui observent (`observerClass`) de demeurer concrètes.

Maintenant que la préoccupation et son adaptateur sont spécifiés, nous allons l'utiliser de la même façon que dans la section 3.3 : *i) rendre* observable les clics de souris sur un bouton et *ii) attribuer à* un label le rôle d'observateur. Un code source possible pour les classes `Button` et `Label` est décrit ci-dessous :

```
103 package application.ihm;
104 import java.awt.event.ActionEvent;
105 import javax.swing.JButton;
106 public class Button extends JButton {
107     public Button(String texte) {
108             super(texte);
109     }
110     protected void fireActionPerformed(ActionEvent event) {
111             super.fireActionPerformed(event);
112     }
113 }
```

```
114 package application.ihm;
115 import javax.swing.JLabel;
116 public class Label extends JLabel {
117     public Label(String texte) {
118             super(texte);
119     }
120 }
```

Ces deux classes appartiennent au paquetage `application.ihm`, tout comme l'adaptateur concret `applicationIHM` qui permet d'intégrer le patron de conception à ces deux classes ; il est décrit en XML par le code suivant :

```
121 <?xml version="1.0" encoding="UTF-8" ?>
122 <jadapt xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
123         xsi:noNamespaceSchemaLocation="JAdaptDOM.xsd">
124   <package>application.ihm</package>
125   <import>designpattern.observer.*</import>
126   <adapter name="ApplicationIHM" extends="ObserverAdapter">
127      <variable name="observableClass" type="Class">
128        Button</variable>
129      <variable name="observerClass" type="Class">Label</variable>
130      <variable name="notifyingMethods" type="Method">
131        fireActionPerformed</variable>
132      <introduce name="observerUpdate">
133        <pointcut>
134           <classes>observerClass</classes>
135        </pointcut>
136        <method>
137          <name>updateObserver</name>
138            <parameters>Observable o</parameters>
139            <body>setText("I have been notified ");</body>
140        </method>
141      </introduce>
142   </adapter>
143 </jadapt>
```

Comme les autres adaptateurs il peut aussi s'exprimer en JAdapt :

```
144 package application.ihm;
145 import designpattern.observer.*;
146 adapter ApplicationIHM extends ObserverAdapter  {
147     Class observableClass = Button;
148     Class observerClass = Label;
149     Method notifyingMethods = fireActionPerformed;
150     introduce(observerUpdate)
151       public void updateObserver(Observable o ) {
152         setText("I have been notified of a button click");
153       }  in  ( observerClass )  ;
154 }
```

L'adaptateur `applicationIHM` hérite (voir ligne 146) de l'adaptateur `ObserverAdapter`, il permet de concrétiser ce dernier pour effectuer les adaptations nécessaires. Les éléments à définir sont les suivants :

– La classe qui est observée, ici c'est `Button` (voir ligne 147).

– La classe qui observe, ici c'est `Label` (voir ligne 148).

– les méthodes de la classe observée qui déclenchent une mise à jour, ici `fire-ActionPerformed` (voir ligne 149).

– la « concrétisation » de l'adaptation `observerUpdate` (voir ligne 150 à 153) afin d'ajouter à la classe qui observe la méthode `updateObserver` qui pour l'instant est vide.

Les classes `Button` et `Label` sont automatiquement modifiées par JAdapt en fonction des adaptations qui viennent d'être décrites.

```
155 package application.ihm;
156 import java.awt.event.ActionEvent;
157 import javax.swing.JButton;
158 import designpattern.observer.*;
159 import java.util.ArrayList;
160 import java.util.Iterator;
161 import java.util.List;
162 public class Button  extends JButton implements Observable{
163   public Button(String texte) { super(texte);  }
164   protected void fireActionPerformed(ActionEvent event)  {
165     fireActionPerformed_Jadapt_0(event);
166     notifyObservers();
167   }
168   public void addObserver(Observer o) {
169     observers.add(o);
170   }
171   public void removeObserver(Observer o) {
172     observers.remove(o);
173   }
174   public void notifyObservers() {
175     for (Iterator iter=observers.iterator();
176          iter.hasNext();)
177     ((Observer)iter.next()).updateObserver(this);
178   }
179   protected List observers=new ArrayList();
180   private void fireActionPerformed_Jadapt_0(ActionEvent event) {
181     super.fireActionPerformed(event);
182   }
183 }
```

```
184 package application.ihm;
185 import javax.swing.JLabel;
186 import designpattern.observer.*;
187 public class Label  extends JLabel implements Observer {
188   public Label(String texte) {
189     super(texte);
190   }
191   public void updateObserver(Observable o) {
192    setText("I have been notified of a button click");
193   }
194 }
```

Ainsi, le code source ci-dessus va se substituer au code source des deux classes correspondantes pendant la phase de compilation. C'est le fait que l'adaptateur `ApplicationIHM` ait défini dans le même paquetage que l'application qui a pour effet de produire une composition *in-situ* (voir ligne 144).

En examinant de plus près le code produit, on constate que les premières adaptations qui sont visibles concernent l'ajout de nouvelles clauses d'importation qui sont rendus nécessaires à cause de plusieurs adaptations. En particulier la modification des liens d'héritage ou d'implémentation d'interface nécessite ces modifications (car les nouvelles interfaces ou la nouvelle super-classe ne sont pas forcement dans le même paquetage que la classe modifiée).

`Button` a le rôle de la classe qui est observée (voir ligne 147) et a subi deux adaptations : la première a eu pour effet d'y fusionner directement la classe `ImplObservable` (voir ligne 84). La classe `Button` contient donc maintenant toutes les méthodes et toutes les variables de la classe `ImplObservable` et elle implémente aussi les mêmes interfaces. La deuxième adaptation (voir ligne 88 à 90) a modifié la méthode `fireActionPerfomed` (voir ligne 149) pour que cette dernière fasse un appel à `notifyObservers()`.

`Label` a le rôle de la classe qui observe (voir ligne 148) et a subi à ce titre deux adaptations : la première (voir ligne 96) ajoute l'interface `Observable` à `Label`, la deuxième (voir lignes 151 à 153) ajoute la méthode `updateObserver` à Label afin que cette dernière ne soit pas abstraite et puisse réagir correctement aux mises à jour des entités qu'elle observe.

L'étude proposée ci-dessus a montré comment JAdapt a permis *i)* d'isoler la préoccupation représentée par un patron de conception et d'encapsuler complètement son protocole de composition dans un adaptateur abstrait et *ii)* de composer à nouveau cette préoccupation avec ses classes clientes dans l'application en utilisant un adaptateur concret. JAdapt a donc à la fois rendu possible la réutilisation de la préoccupation et offert un protocole de composition pour guider l'utilisateur à implémenter d'un adaptateur abstrait.

7.2.3.2 Bilan relatif aux patrons de conception

L'étude relative à la réutilisation des patrons de conception qui a été conduite dans la section 3.3 a montré que les approches par objets (avec ou sans métaprogrammation et avec ou sans généricité) ne permet pas de séparer l'implémentation des patrons de conception de leurs classes clientes. De même, si la programmation par aspects permet de séparer les patrons de conception du reste de l'application, elle ne permet pas de les réutiliser aussi facilement qu'on le souhaiterait. En effet la programmation par aspects ne permet pas d'encapsuler de manière abstraite et complète les protocoles de composition. La programmation par sujets permet de séparer les patrons de conception de l'application mais l'impossibilité de décrire un protocole de composition rend leur réutilisation extrêmement difficile.

En conclusion, les approches étudiées dans la section 3.3 ne permettent pas de réutiliser efficacement des patrons de conception, car aucune de ces approches n'offre une encapsulation complète des protocoles de composition par manque de capacité d'abstraction. Par contre, la section précédente montre que JAdapt offre cette facilité et améliore de ce fait les capacités de réutilisation.

7.3 SYNTHESE DES RESULTATS

L'étude relative à la réutilisation des préoccupations dans le monde des objets est maintenant complète. Ainsi, une étude détaillée de l'état de l'art a mis en évidence les aspects importants dont il faut tenir compte et la capacité des principales approches à les supporter (voir chapitre 3). Le présent chapitre a permis quand à lui, d'évaluer l'implémentation proposée au chapitre 6 pour le modèle défini au chapitre 5, avec les mêmes critères que ceux utilisés pour les autres approches de l'état de l'art. Plusieurs idées fortes ressortent de cette étude ; nous allons les passer en revue dans les lignes qui suivent.

Une fois les préoccupations séparées, c'est-à-dire après l'élimination des couplages qui peuvent exister entre elles, la principale difficulté réside dans la composition qui s'avère nécessaire pour reformer l'application. Cette composition suit une démarche qui s'appuie sur des adaptations et elle est réalisée la plupart du temps par les approches étudiées avant la compilation.

La capacité à supporter différents types d'adaptations et différentes manières de les encapsuler détermine directement la facilité avec laquelle l'utilisateur va pouvoir composer des préoccupations. Nous avons appliqué ces deux critères aux approches de l'état de l'art dans la section 3.4 et nous en avons tiré une synthèse. Afin de la compléter, nous nous y intéressons à nouveau et nous lui intégrons les résultats qui correspondent à JAdapt.

Pour éviter d'avoir à encapsuler dans la même entité toutes les adaptations nécessaires pour reformer une application, les adaptations doivent être scindées en éléments plus petits mieux appropriés à la modélisation de compositions des préoccupations. Ces éléments plus petits ne portent que sur des sous-ensembles de préoccupations à composer. Dans notre implémentation ces éléments s'appellent des adaptateurs, ils sont comparables aux aspects ou au sujets d'autres modèles vus dans le chapitre 3.

De plus, pour simplifier l'utilisation d'une préoccupation, un des moyens les plus intéressants est d'utiliser un protocole de composition. Ce protocole de composition doit être implémenté en même temps que l'implémentation de la préoccupation et doit décrire de manière suffisamment abstraite[1] les adaptations qui sont nécessaires à sa réutilisation. Il devra pouvoir ensuite être spécialisé et concrétisé afin d'être complète et de s'adapter au contexte d'une application donnée. L'utilisation de la préoccupation passe donc ensuite par la spécialisation par héritage de son protocole de composition.

Le tableau suivant synthétise les capacités d'encapsulation et d'abstraction des protocoles de composition de toutes les approches étudiées.

Paradigme/Modèle	Encapsulation	Abstraction
Objet	Non	Non
AspectJ	Incomplète	Incomplète
Hyper/J	Non	Non
JAdapt	Oui	Oui

Tableau 10. *Encapsulation et abstraction des protocoles de composition dans les approches étudiées*

Notre approche est la seule à pouvoir réellement encapsuler et abstraire les adaptations nécessaires à la réutilisation d'une préoccupation dans un protocole de composition. Ce résultat a été rendu possible car notre approche a été conçue dans ce but précis en s'appuyant sur les contributions d'Hyper/J et d'AspectJ. AspectJ montre des résultats incomplets dans ce domaine car ce modèle et l'implémentation associée ne sont pas capables de traiter les types d'adaptation basée sur la fusion.

Après les capacités d'abstraction et d'encapsulation des adaptations il est utile de s'intéresser aux types d'adaptation qui sont proposées par chacune des approches. Ils sont recensés dans le tableau suivant[2].

[1] Pour être indépendant des applications.
[2] Ils s'appuient bien évidemment sur les résultats du Chapitre 3.

Description	JAdapt	AspectJ	Hyper/J
Implémentation de nouvelles interfaces	Oui	Oui	Oui
Fusion de classes et de méthodes	Oui	Non	Oui
Ajout ou redéfinition de méthodes	Oui	Oui	Oui
Ajout de nouvelles variables d'instance (et de classe)	Oui	Oui	Oui
Interception avant, après, autour, sur exception des méthodes	Oui	Oui	Partiellement
Interception des accès aux variables d'instance (et de classe)	Non implémenté	Oui	Non

Tableau 11. *Support des différents types d'adaptation pour les approches étudiées*

Les résultats du tableau 11 ne mentionnent pas le paradigme de l'objet car ce dernier ne possède aucune forme de composition de préoccupations autre que l'héritage et l'agrégation. A l'énoncé de ces résultats on comprend pourquoi AspectJ ou Hyper/J n'ont pas permis de composer toutes les préoccupations étudiées. En effet, ils ne permettent pas de réaliser toutes les adaptations nécessaires à la réutilisation des préoccupations. Comme notre approche a été construite à partir du recensement des limitations des approches de l'état de l'art, il est naturel qu'elle prenne effectivement en compte tous les types d'adaptations. L'interception d'accès aux variables n'a pas été encore implémentée mais le modèle permet d'envisager son support. On peut remarquer qu'en Java la méthodologie couramment admise incite à toujours passer par des accesseurs et des modificateurs (qui eux sont des méthodes). Cela rend moins important le fait de supporter ou non l'interception d'accès aux variables d'instance.

Pour être complet il faut faire l'inventaire pour chaque approche des capacités d'abstraction des différentes adaptations, en d'autre termes, des possibilités offertes pour définir les adaptations indépendamment du contexte et donc les décrire de manière abstraite. Les résultats montrent que tous les paradigmes ne sont pas capables d'abstraire toutes les adaptations qu'ils supportent. Ceci permet de mieux expliquer les résultats du **tableau 10**.

Description	JAdapt	AspectJ	Hyper/J
Implémentation de nouvelles interfaces	Oui	Non	Non
Fusion de classes et de méthodes	Oui	Non	
Ajout ou redéfinition des méthodes	Oui	Non	
Ajout de nouvelles variables d'instance (et de classe)	Oui	Non	
Interception avant, après, autour, sur exception des méthodes	Oui	Oui	
Interception des accès aux variables d'instance (et de classe)	Non Implémenté	Oui	

Tableau 12. *Capacité d'abstraction des adaptations*

Les résultats du tableau 12 rappellent qu'Hyper/J ne possède aucune relation d'héritage sur ses modules de composition et donc qu'il ne permet ni une description des adaptations indépendamment du contexte ni la spécification d'un protocole de composition abstrait. A propos d'AspectJ, ce dernier ne possède des capacités d'abstraction que pour certaines adaptations ; ceci est parfaitement cohérent avec le fait que sa capacité à implémenter un protocole de composition est limitée (voir tableau 10). JAdapt de son coté offre des capacités d'abstraction complètes[1], ce qui explique pourquoi JAdapt est la seule approche qui a été capable d'encapsuler correctement tous les protocoles de composition.

Abstraire des adaptations ou les rendre indépendantes du contexte d'utilisation revient la plupart du temps à abstraire la cible des adaptations. Nous récapitulons pour les différentes approches leur capacité à décrire des cibles abstraites.

Description	JAdapt	AspectJ	Hyper/J
Classe ou ensemble de classes	Oui	Non	Non
Méthode ou ensemble de méthodes	Oui	Oui	Non
Variable d'instance ou ensemble de variables d'instance	Non Implémenté	Non	Non

Tableau 13. *Possibilités d'abstraction des cibles des adaptations*

Les résultats contenus dans le tableau 13 expliquent pourquoi AspectJ ne permet pas d'abstraire les adaptations qui portent sur des cibles représentant des classes, En effet ce dernier ne permet pas d'implémenter des points de jointure abstraits vers des classes. Hyper/J quant à lui ne possède aucune capacité de description de cible abstraite. Enfin, JAdapt offre des moyens d'abstraction pour toutes les cibles d'adaptation à l'exception des variables d'instance qui ne sont pas intégrées dans la version actuelle de l'implémentation.

La dernière comparaison qui reste à évoquer porte sur la nature de la composition. Alors que deux des trois exemples (les préoccupations non fonctionnelles et les patrons de conception) ont nécessité d'avoir une composition *in-situ* qui modifie directement les classes concernées, l'exemple des préoccupations fonctionnelles nécessite pour sa part d'avoir une composition *ex-situ* qui produit de nouvelles classes.

Description	JA-DAPT	AspectJ	Hyper/J
Composition in-situ	Oui	Oui	Non
Composition ex-situ	Oui	Non	Oui

Tableau 14. *Support des différents types de composition.*

Les résultats du tableau 14 expliquent pourquoi AspectJ n'as pas permis de séparer et de composer les préoccupations fonctionnelles. De même, ils expliquent pourquoi Hyper/J n'a pas permis de le faire sur les deux autres exemples. JAdapt réconcilie les deux approches car il supporte les deux natures de composition.

[1] A part pour les interception d'accès aux variables et ceci pour les raisons précédemment expliquées.

7.4 CONCLUSION

Nous venons de voir dans ce chapitre comment utiliser JAdapt (chapitre 6) une implémentation de notre modèle (chapitre 5) pour le langage Java. Elle a été développée comme un plug-in de l'environnement de développement Eclipse [ECL 04].

Pour montrer l'utilisation de notre implémentation et pour valider ses capacités de réutilisation nous avons repris les trois exemples (chapitre 3) qui nous on permis d'évaluer l'état de l'art. L'évaluation des capacités de réutilisation correspond à la facilité de réutilisation des préoccupations et aux capacités d'adaptation disponibles.

Le modèle sous-jacent à notre implémentation a été défini en s'appuyant sur les limitations mais aussi sur les apports des différentes approches étudiées dans le chapitre 3. Ce chapitre valide l'implémentation proposée pour notre modèle.

Dans un premier temps nous avons montré comment séparer une préoccupation non fonctionnelle (un mécanisme de trace). JAdapt a permis d'implémenter le protocole de composition de cette préoccupation sous la forme d'un adaptateur abstrait. La réutilisation de la préoccupation à été facile grâce au guide offert par l'adaptateur abstrait qu'il a suffit de concrétiser par héritage. La section 3.1 a montré que la seule approche capable de faire aussi bien est la programmation par aspects par l'intermédiaire d'AspectJ.

Dans un deuxième temps, nous avons étudié la séparation de trois préoccupations fonctionnelles : la structure de l'arbre des expressions arithmétique, l'évaluation d'expressions arithmétiques, et l'affichage de structures arborescentes. Parmi ces trois préoccupations, seule la dernière nécessite un protocole de composition, car c'est la seule qui doit être composée pour être utilisable. Ce protocole de composition a pu être encapsulé dans un adaptateur abstrait par JAdapt. Les trois préoccupations ont pu être composées ensemble par un adaptateur concret qui : *i)* réutilise le protocole de composition de la troisième préoccupation pour faciliter la composition de cette dernière, *ii)* produit un nouvel ensemble de classes représentant le résultat de la composition pour permettre la réutilisation des trois préoccupations à d'autres endroits dans la même application, *iii)* ne modifie aucune des trois préoccupations pour les composer ensemble. La section 3.2 a permis en particulier de constater que seule la programmation par sujets avait permis de composer ces trois préoccupations ensemble mais sans pouvoir encapsuler et donc utiliser le protocole de composition de la troisième préoccupation et en nécessitant l'utilisation de classes de transition entre les préoccupations pour pouvoir les composer correctement (en effet, Hyper/J ne supporte pas toutes les adaptations nécessaires).

Dans un dernier temps nous avons proposé une séparation du patron de conception de l'observateur. Le protocole de composition de ce patron de conception a été entièrement encapsulé dans un adaptateur abstrait par JAdapt. Cet adaptateur abstrait a été réutilisé par héritage pour composer la préoccupation avec des classes clientes. Ceci a permis de faciliter et de guider la composition, en simplifiant ainsi le travail de l'utilisateur. La section 3.3 a montré que pour le même exemple c'est la programmation par aspect qui s'en sort le mieux par rapport aux autres approches. Cependant AspectJ n'a permis d'encapsuler qu'une partie du protocole de composition de la préoccupation. La réutilisation a donc nécessité plus de travail de la part de l'utilisateur qui a été par la même occasion moins guidé dans sa démarche.

Il semble clair au regard des résultats mis en évidence que JAdapt est la seule approche qui a permis d'avoir les deux propriétés qui nous semblent fondamentales : la facilité de réutilisation et la capacité d'adaptation. Ceci est vrai quel que soit l'exemple considéré. De plus, comme nos exemples couvrent un domaine assez vaste de besoins de réutilisation des préoccupations dans le monde de l'objet, nous pouvons conclure sur le fait que JAdapt a atteint les objectifs fixés lors de la définition de notre modèle de séparation et de composition des préoccupations.

Chapitre 8

Perspectives

A partir des travaux présentés dans ce mémoire de thèse, nous présentons ci-dessous différents axes de recherche à approfondir. Il s'agit à cours terme de l'amélioration de l'implémentation de JAdapt, et à moyen terme de l'extension de notre modèle en vue de l'utiliser dans le paradigme de la programmation orientée composant puis pour les développements dirigés par les modèles.

8.1 AMELIORATION DE JADAPT

Il est nécessaire d'améliorer l'implantation de notre modèle. Ces améliorations concernent le cœur de l'implémentation mais aussi son intégration dans l'environnement de programmation d'Eclipse.

A propos du moteur de composition des préoccupations de Jadapt, on constate que l'implémentation de notre modèle n'est pas complète. En effet, à l'heure actuelle JAdapt ne supporte pas des adaptations pour l'interception d'accès aux variables. Dans un premier temps cette adaptation n'avait pas été jugée nécessaire car les habitudes de programmation en Java suggèrent l'utilisation d'accesseurs en lecture et en écriture pour les variables d'instances, et que ces accesseurs sont des méthodes ; à ce titre elles peuvent donc être interceptées par JAdapt. Cependant comme rien n'oblige le programmeur Java à utiliser des accesseurs pour consulter ou modifier le contenu d'une variable, il semble intéressant d'offrir directement ce type d'adaptation.

L'implémentation actuelle des interceptions de méthodes dans JAdapt introduit une indirection à chaque interception effectuée sur une méthode. Ceci a permis à la fois un développement plus rapide du prototype et de simplifier la composition de ce type d'adaptation. Par contre, cette indirection à un coût supplémentaire à savoir un appel de méthode en plus pour chaque méthode interceptée. Bien que dans certains cas d'optimisation les techniques liées au « just-in time compiling » intégrées à la machine virtuelle Java puissent annuler cette indirection, il est nécessaire d'agir plus globalement pour supprimer ce coût. Pour cela il faut faire évoluer l'implémentation des interceptions d'appels de méthodes afin de ne plus procéder par indirection mais de modifier directement le code de la méthode interceptée. Ainsi le coût induit par l'adaptation sera nul, car le code qui est ajouté par l'interception correspond directement au code que l'utilisateur aurait écrit pour utiliser la préoccupation.

Le moteur de composition de JAdapt ne permet pas de faire de la compilation incrémentale. Cette limitation à un impact très important sur les temps de compilation et sur la manière d'utiliser Eclipse. En effet Eclipse est prévu pour compiler de manière incrémentale chaque fichier source à chaque enregistrement de celui-ci, ce qui a pour effet d'annuler les adaptations qui ont déjà été réalisées sur ce fichier. Dans la version actuelle de l'implémentation aucun mécanisme n'est prévu pour prendre en compte l'impact d'une modification sur les adaptations exécutées au préalable et il faut donc déclencher une recompilation totale pour obtenir à nouveau un système cohérent. Un examen approfondi des outils offerts par Eclipse montre que ce dernier fournit l'ensemble des éléments nécessaires à la mise en oeuvre d'une compilation incrémentale : *i)* des événements qui peuvent être interceptés par les plug-ins pour signaler l'enregistrement des fichiers et le lancement de l'application et *ii)* une réification pour chacun des fichiers manipulés par la plate-forme, des

différences qui existent entre deux versions d'un fichier créés par deux enregistrements successifs[1]. Il devient donc possible de faire de la composition de préoccupations incrémentale et ceci est un objectif à court terme.

Le système de points de jointure de JAdapt peut être étendu afin de se rapprocher des possibilités offertes par les systèmes de points de jointure de la programmation par aspects (voir 2.3.4 et 2.4.1). Ceci concerne essentiellement les points de jointure qui portent sur les méthodes car eux seuls concernent la programmation par aspects. Par rapport à ces points de jointure, nous en avons implémentés un ensemble réduit qui permet simplement de composer convenablement les préoccupations. Beaucoup reste à faire dans ce domaine, par exemple il sera intéressant de s'intéresser : *i)* aux points de jointure qui permettent de considérer l'information de la pile d'appels, ou *ii)* à une meilleure prise en charge des expressions régulières. Ces deux derniers points sont particulièrement bien traités par AspectJ, de plus [DMS 01] ou [DL 02] montre qu'un système de points de jointure peut être amélioré pour certains cas d'utilisation. Il faudra cependant faire attention à n'offrir que ce qui est nécessaire à une réutilisation des préoccupations ; l'objectif n'est pas de concurrencer AspectJ ou tout autre langage généraliste pour la séparation des préoccupations.

Il est bien sûr possible d'envisager d'autres améliorations du cœur de JAdapt, mais nous pensons que celles qui ont été présentées sont les plus importantes. Par contre beaucoup de choses restent à faire pour aboutir à une intégration de JAdapt qui soit plus dans l'esprit de la plate-forme de développement Eclipse. Nous examinons dans les lignes qui suivent comment améliorer cette intégration.

En premier lieu, un des aspects qui nous parait le plus important à considérer est la vérification de la compatibilité de notre implémentation avec la version 3 de la plate-forme. Nous n'avons pas pu nous en assurer pour le moment car cette version est encore en plein développement et donc elle est à la fois trop sujette aux changements et susceptible de contenir de nombreux *bugs*. Nous avons jugé préférable d'attendre qu'elle se stabilise.

JAdapt ne bénéficie que d'une intégration minimale dans la plate-forme. Cette intégration est purement fonctionnelle et permet uniquement de lancer notre précompilateur avant le compilateur natif pour Java.

Cette intégration peut être enrichie par la prise en compte des différentes vues auxquelles l'utilisateur a accès pour manipuler ses projets. Dans la perspective Java d'Eclipse par exemple, une classe qui est en cours d'édition dispose d'une vue qui affiche uniquement sa structure générale. Cette facilité peut être appliquée aux fichiers XML qui contiennent les adaptateurs.

De même, comme c'est le cas dans le plug-in d'AspectJ pour Eclipse, il est intéressant de signaler chaque cible d'un point de jointure directement dans l'environnement d'Eclipse. Chaque méthode interceptée par un point de jointure (qui est donc la cible directe ou indirecte d'une adaptation) pourrait porter une petite décoration signalant la source de cette adaptation. C'est un moyen de vérification assez sommaire mais qui donne une idée précise et visuelle des cibles des adaptations.

Les adaptateurs sont actuellement implémentés uniquement à travers une description en XML. Nous avons proposé une projection de la description en XML vers un langage dédié à la manipulation des préoccupations. La syntaxe associée à ce langage a été utilisée dans ce document afin de montrer une représentation plus intuitive des exemples mis en oeuvre. Cependant la version actuelle de JAdapt ne supporte pas la manipulation d'adaptateurs décrits dans ce langage. Si le fait d'implanter les adaptations à partir de fichiers XML est particulièrement intéressants pour à la fois pérenniser l'implémentation et faciliter son partage, il est aussi important de fournir des transformations bidirectionnelles entre les deux représentations. A ce propos Eclipse fournit tous les outils

[1] Notons au passage que ce mécanisme de base participe à la mise en œuvre d'un système intégré de versions des fichiers

nécessaires à la réalisation d'analyses lexicales et syntaxiques et ses éditeurs possèdent des fonctionnalités pour la mise en forme, la coloration syntaxique et la complétion de code source[1] .

Maintenant que les améliorations relatives à l'implémentation du modèle ont été considérées, nous allons nous intéresser au modèle proprement dit et en particulier à sa faculté d'adresser d'autres paradigmes que l'objet comme les composants ou les développements dirigés par les modèles (MDA ou DDD).

8.2 PROGRAMMATION ORIENTEE COMPOSANTS

La programmation orientée composants permet de construire une application à partir d'un ensemble de composants. Un composant est un bloc de construction réutilisable dans différents contextes ou applications. Ces blocs de construction forment une application à travers un processus compositionnel appelé intégration [Szy 98]. L'intégration de composants pour former une application peut être réalisée par une plate-forme à composants, un langage de description d'architecture, etc.

En pratique, cette phase d'intégration comporte encore de nombreuses limitations qui semblent essentiellement être attribuées aux langages et aux environnements de programmation actuels. Ceux-ci reposent sur l'hypothèse que les composants sont compatibles entre eux [Bos 97] alors qu'un aspect séduisant de la programmation orientée composants est justement de pouvoir utiliser des « composants sur l'étagère ». Ces composants possèdent la particularité d'être développés par des tiers sans concertation et ils sont donc potentiellement non compatibles.

Cette section montre notamment que ces problèmes d'intégration proviennent de la faiblesse des mécanismes d'adaptation fournis par les langages et les plates-formes à composants, ce qui rend plus complexe l'utilisation de composants sur l'étagère. C'est l'intégration de ce type de composant qui va servir de cadre à une perspective d'utilisation du modèle de séparation et de composition de préoccupations présentée dans le chapitre 5.

Un composant développé isolément des autres se doit d'avoir un espace de type complet sans référence extérieure. Néanmoins pour être réutilisable dans différents contextes un composant doit avoir une granularité qui lui permet de s'intégrer dans des systèmes variés. Pour cela un composant doit déclarer les services requis pour son fonctionnement et les services qu'il fournit aux autres composants ; en d'autres termes il doit être autosuffisant du point de vue de ses spécifications mais pas de celui de son fonctionnement.

Ce besoin d'autosuffisance a déjà été exposé lors du bilan sur la réutilisation des préoccupations dans le monde de l'objet (voir section 3.5). Pour une préoccupation dans le monde de l'objet cette autosuffisance correspond à plusieurs caractéristiques : *i)* limiter au maximum le couplage entre la préoccupation et le reste de l'application, *ii)* pouvoir implémenter et encapsuler tout le protocole de composition de la préoccupation et *iii)* pouvoir spécialiser par héritage ce protocole de composition. Ces caractéristiques permettent de faciliter et de guider la réutilisation des préoccupations.

La suite est organisée en quatre sous-parties. La section 8.2.1 présente le contexte de l'étude. La section 8.2.2 montre un exemple de composant sur l'étagère. La section 8.2.3 étudie l'intégration de tels composants dans une application. La dernière section propose un bilan.

[1] Ces dernières permettraient de faciliter l'écriture des adaptateurs pour l'utilisateur.

8.2.1 Hypothèse de travail pour les composants sur l'étagère

Cette étude est basée sur cinq hypothèses de travail qui nous semblent particulièrement correspondre aux besoins de l'industrie :

– les applications doivent pouvoir utiliser majoritairement des composants existants ; cela répond à un besoin de productivité ;

– les composants sont conçus indépendamment par différents acteurs, et ils sont le plus souvent développés et vendus par des industriels ;

– le code source d'un composant ne doit ou ne peut pas être modifié. Soit il n'est pas disponible, soit il pourrait l'être mais ceci n'est pas souhaitable car les mises à jour ultérieures des composants seraient rendues plus difficiles. Concernant ce dernier point nous ne sommes pas favorables à des modifications directes du code source (ou même du code binaire) d'un composant. Cependant, une génération automatique de ces modifications, à partir d'un ensemble de déclarations visant à adapter le composant, serait acceptable car il suffirait alors de mettre à jour ces déclarations et non le code source du composant pour l'adapter ;

– les composants sont écrits avec des langages statiquement typés. Nous privilégions l'utilisation de ces langages car ils répondent en général mieux au besoin de qualité des logiciels et parce qu'ils sont plus largement diffusés. Néanmoins, certains des problèmes que nous allons évoquer s'appliquent aussi aux langages dynamiquement typés, mais ils ne seront pas ici évoqués de manière plus détaillée ;

– un composant est conforme à la définition donnée par [Szy 98] : « *un composant logiciel peut être vu comme une unité de composition avec des interfaces fournies et requises contractuellement spécifiées. Cette unité est déployable et toutes ses dépendances externes sont explicites.* » Un composant est une boîte noire qui publie des interfaces[1] fournies et requises.

L'exemple de développement d'un composant sur l'étagère proposé dans la partie suivante s'appuie sur ces hypothèses.

8.2.2 Exemple de composant sur l'étagère

Un composant sur l'étagère doit donc être conçu dans un monde fermé où seules des interfaces fournies et requises servent de liaison avec le monde extérieur. Le cadre d'utilisation de ces composants empêche en effet ceux qui les développent de faire des hypothèses sur les applications dans lesquelles le composant sera intégré où sur les autres composants qui, d'une part, participent à l'intégration et, d'autre part, peuvent avoir des provenances différentes. Ceci correspond à l'étape de séparation des préoccupations du monde de l'objet.

Les interfaces requises et fournies contiennent des méthodes qui font, à leur tour, référence à des types complexes (autre que les types de base comme les chaînes de caractères ou les nombres par exemple) pour leurs paramètres ou leur type de retour. Pour garantir la complétude du typage (l'indépendance) du composant, celui-ci doit donc contenir la déclaration des types des paramètres des méthodes.

On définira donc un composant sur l'étagère comme une préoccupation indépendante du contexte qui contient non seulement les interfaces requises et fournies mais aussi une description de tous les types auxquels ces interfaces font référence. En particulier les fonctionnalités qu'exporte le composant vers l'extérieur ne feront aucune référence à un type externe (qui n'est donc pas défini dans le composant).

[1] Une interface est un ensemble de méthodes avec leurs signatures.

Les types utilisés par un composant peuvent eux-mêmes être déclarés par des interfaces que nous qualifierons d'interfaces « utilisées » (car elles sont nécessaires à l'espace de type du composant). Pour respecter le typage statique du composant (avec son implémentation), les interfaces représentant les types « utilisés » doivent contenir aussi toutes les méthodes nécessaires au composant lui-même, sans faire d'hypothèse sur les fonctionnalités qui pourraient être utiles à d'autres composants et qui permettraient de rendre le type plus général.

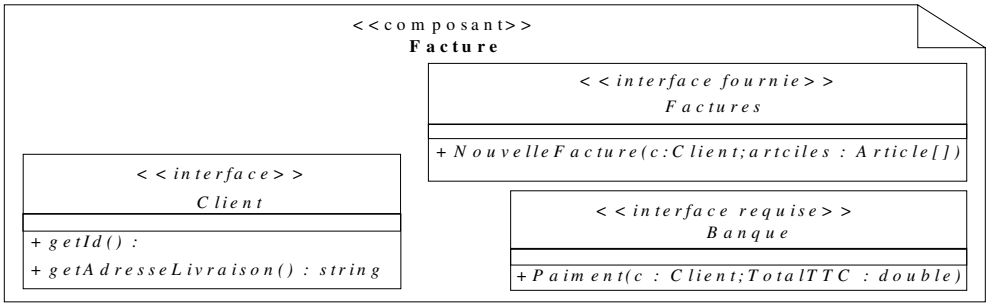


Figure 21, *Le Composant **Facture***.

Pour illustrer notre propos, un composant permettant de traiter les factures est spécifié dans la figure 21 ; il est conçu en respectant les hypothèses mentionnées dans la section 8.2.1.

Ce composant (**`Facture`**) traite les factures : il fournit des fonctionnalités pour créer une facture à partir d'un client et d'un ensemble d'articles (pour ne pas surcharger le schéma, le composant représentant l'interface `Article` n'apparaît pas). Le composant **`Facture`** requiert une interface de type `Banque` permettant un paiement automatique de la facture dès la création.

Comme l'interface fournie `Factures` possède une méthode faisant référence au type `Client`, le composant contient une déclaration de ce type sous la forme d'une interface. Cette interface contient uniquement les méthodes du type `Client` utilisées par le composant.

L'interface requise `Banque` utilise aussi le type `Client`, mais comme le composant est créé indépendamment du contexte d'utilisation, nous ne pouvons faire aucune hypothèse sur les besoins du composant qui fournira cette interface en relation au type `Client` en particulier l'interface `Client` du composant **`Facture`** n'a pas vocation à contenir de méthodes pouvant éventuellement être utilisées par le composant qui fournira l'interface `Banque` et vice versa.

Le composant **`Facture`** a donc été spécifié avec des types `Client` et `Banque` qui couvrent ses propres besoins. Dans la section suivante le composant va être intégré avec un composant **`Client`** et un composant **`Banque`**, tous deux trouvés sur l'étagère.

8.2.3 Intégration de composants sur l'étagère

Une intégration de composants sur l'étagère satisfaisant aux contraintes définies dans la section 8.2.1 est décrite dans la figure 22 ; elle intègre les composants **`Facture`**, **`Client`** et **`Banque`** pour former une application. Pour tester l'intégration, l'interface fournie du composant **`Facture`** propose la méthode `NouvelleFacture` qui peut tenir lieu de point d'entrée de l'application.

Ces trois composants ne sont que superficiellement incompatibles car les fonctionnalités désirées sont présentes : le composant **`Banque`** fournit le service requis par le composant **`Facture`** et l'interface `Client` du composant **`Client`** contient toutes les fonctionnalités nécessaires aux composants **`Facture`** et **`Banque`**.

L'incompatibilité entre ces trois composants vient de la triple déclaration de l'interface `Client` qui est nécessaire pour satisfaire à leur statut de composant sur l'étagère. L'assemblage de ces trois composants nécessite donc la fusion de ces trois interfaces. Ce problème est proche de celui abordé par [MS 97], néanmoins nous proposons une solution différente car [MS 97] ne prend pas en compte les spécificités des composants sur l'étagère.

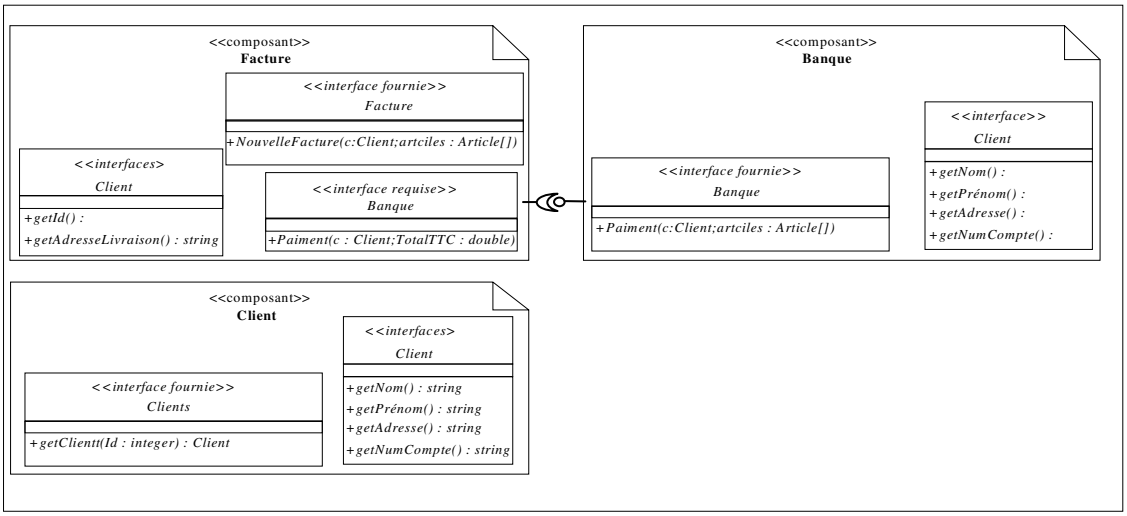


Figure 22, *Intégration de composants sur l'étagère*

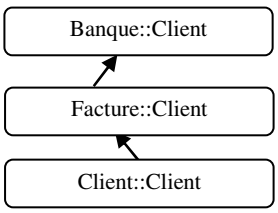
Notons qu'un moyen de vérifier l'intégration des composants serait d'utiliser la programmation par contrat comme le montre [HLS 97].

Le problème des interfaces déclarées plusieurs fois peut se résoudre par adaptation en créant des liens d'héritage entre ces différentes interfaces pour les rendre compatibles entre elles comme le propose l'algorithme ci-dessous.

```
Id = Une interface présente dans plusieurs composants (ici Client)
E  = Ensemble de composants qui déclarent Id (ici Facture, Client, Banque)
F  = Composant qui fournit le service représenté par Id

Pour tout C : composant dans (E–F)
  E' = Ensemble des composants fournissant des services à C qui utilisent Id
  Pour tout C'' : composant dans (E'–F)
      Modification de C::Id pour qu'elle hérite de C''::Id
  Modification de F::Id pour qu'elle hérite de C::Id
```

L'utilisation de cet algorithme suivie d'une simplification des liens d'héritage[1] permet d'obtenir le graphe d'héritage ci–contre. Le composant ***Client*** doit ensuite être adapté pour refléter le nouvel arbre d'héritage de l'interface `Client`. Dans notre exemple, le composant ***Client*** doit maintenant implémenter la méthode `getAdresseLivraison` définie par l'interface `Client` du composant ***Facture***.



[1] Cet algorithme relativement simpliste n'est pas développé ici dans un souci de concision.

Actuellement les plates-formes à composants et les langages d'architecture n'offrent pas les fonctionnalités nécessaires pour adapter les trois composants. Nous proposons d'utiliser notre modèle de composition à travers son implémentation JAdapt.

Nous allons utiliser un adaptateur pour modifier des relations d'héritage entre les différentes interfaces `Client` (lignes 2 et 3) et pour modifier l'implémentation du composant ***Client*** (lignes 5 à 7). Ceci évite une modification invasive de ***Client*** et de son interface `Client` pour implémenter la méthode `getAdresseLivraison` qui manque à cette dernière. Ces modifications seront automatiquement reportées sur la classe du composant ***Client*** qui implémente l'interface `Client`. L'adaptateur qui réalise ces adaptations de manière modulaire est décrit ci-dessous :

```
1   adapter Integration_BanqueClientFacture {
2     implements Banque.Client  in Facture.Client;
3     implements Facture.Client in Client.Client ;
4
5     introduce String getAdresseLivraison() {
6       return getNom()+ « » + getPrénom() + « » + getAdresse() ;
7     } in Client.Client;
8   }
```

Notre modèle de composition permet de résoudre les problèmes d'incompatibilité des composants sur l'étagère grâce aux différentes adaptations qu'il supporte. Ceux-ci peuvent ensuite être intégrés grâce à des plates-formes à composants pour construire une application. Comme les adaptations sont décrites en dehors du composant, l'évolution d'un composant peut se faire indépendamment des autres composants, pourvu qu'il garde les mêmes interfaces. En cas d'évolution sur les interfaces : soit l'adaptateur doit être mis à jour (si les interfaces dupliquées sont modifiées), soit c'est l'intégration qui doit l'être (si les interfaces requises ou fournies sont modifiées).

8.2.4 Conclusion

Les composants sur l'étagère ne peuvent, ni dans leur spécification, ni dans leur implémentation, faire d'hypothèses sur leurs futurs contextes d'utilisation. Nous avons montré que cette indépendance par rapport au contexte oblige les composants sur l'étagère à encapsuler non seulement les interfaces requises et fournies mais aussi tous les types auxquels ces interfaces font référence. Ces besoins correspondent à l'application de la séparation des préoccupations [LH 95] dans le paradigme des composants.

Nous avons montré que les composants sur l'étagère reportent les éventuelles redondances de type et leurs incompatibilités à la phase d'intégration. Il n'est pas réaliste d'envisager l'existence d'un ensemble à la fois exhaustif et standardisé des types généraux qui pourraient être utilisés pour concevoir les composants. Nous proposons donc de réutiliser notre travail sur JAdapt pour permette d'adapter des composants. Pour y parvenir nous avons utilisé des adaptations comme la modification des liens d'héritage entre les différents types et l'introduction de nouvelles méthodes.

Comme les plates-formes à composant actuelles ne permettent pas de faire les adaptations supportées par JAdapt, ces dernières doivent à leur tour être intégrées dans une plate-forme à composants. Ceci nécessite de projeter notre modèle de séparation et de composition des préoccupations dans le paradigme des composants.

Cette perspective montre que des incompatibilités de type peuvent survenir lors de la conception et l'intégration de composant. Ces problèmes de typage nécessitent des capacités d'adaptation non prévues aussi bien dans les langages que dans les plates-formes à composants. Pourtant la conception de composants sur l'étagère nécessite de faire face à ces problèmes. Nous pensons que ce mode de conception des composants est nécessaire pour améliorer l'utilisation des composants dans le milieu industriel car elle répond à des problèmes de productivité. Nous désirons enrichir le

modèle proposé afin de l'adapter aux spécificités du paradigme des composants, pour par exemple l'implémenter dans un système ouvert comme [BCS 02] [CLB 01].

8.3 MDA

MDA [MDA 04], « Model Driven Architecture » ou encore architecture dirigée par les modèles est une approche qui permet d'abstraire davantage la conception d'application de son implémentation. Pour y parvenir, elle propose d'utiliser UML [UML 04] et de considérer deux points de vue d'un modèle métier : *i)* le PIM, « Platform Independent Model » ou modèle indépendant de la plate-forme et *ii)* le PSM, « Platform Specific Model » ou modèle spécifique à la plate-forme.

La notion de PIM permet de s'intéresser à la conception de l'application sans se soucier de la plate-forme d'implémentation. Ceci permet d'utiliser toutes les possibilités d'UML qui ne sont pas forcement présentes dans les langages de programmation ou les plates-formes à composants comme par exemple des classifieurs autres que les classes ou les interfaces ou bien encore des relations différentes de l'héritage entre classes.

Un PSM se déduit du PIM par la prise en compte des spécificités et des limites de la plate-forme ou du langage utilisé pour implémenter l'application. De manière idéale le PSM doit aussi pouvoir se projeter directement vers du code source compréhensible par la plate-forme d'implémentation.

L'approche MDA favorise donc la conception d'applications à travers l'abstraction des spécificités de la plate-forme d'implémentation décrite dans le PIM et par la prise en compte de ces spécificités par le PSM. La principale difficulté de l'approche MDA est donc la transformation de modèle entre les PIM et PSM.

Une transformation de modèle réalisée de manière programmative nécessite un langage capable de manipuler les entités réifiées du modèle. La réification de ces entités est facilitée par l'existence du MOF ; ce dernier réifie toutes les entités d'UML et donc toutes les descriptions réalisables en UML. La manipulation des entités nécessite d'être capable de modifier toutes leurs propriétés et de pouvoir en créer de nouvelles dynamiquement. La transformation de PIM vers PSM correspond souvent à une adaptation du PIM afin de prendre en compte les limitations de la plate-forme.

Notre modèle a été conçu pour le paradigme de l'objet, il permet en particulier de manipuler deux types d'entité : les classes et les ensembles de classes. Il est pourtant envisageable d'enrichir notre modèle pour qu'il puisse manipuler toutes les entités disponibles en UML. Ceci permettrait d'utiliser notre modèle comme base pour faire de l'adaptation de modèle UML, pour implémenter un nouveau langage pour faire de la transformation de modèle en UML.

Dans cette optique il est aussi important de pouvoir abstraire les adaptations nécessaires aux transformations PIM vers PSM car, même si ces dernières sont spécifiques au PIM et à la plate-forme d'implémentation, il doit être possible de décrire des transformations plus génériques qui ne seraient pas spécifiques à la plate-forme d'implémentation. Ceci permettrait de simplifier la transformation PIM vers PSM car elle ne nécessiterait qu'une spécialisation d'un adaptateur pour un PIM pour une plate-forme donnée.

De même, mais à un autre niveau, notre modèle pourrait être aussi utilisé pour séparer les différentes préoccupations prises en compte au niveau des PIM. Ces derniers représentent l'application, ils contiennent donc toutes les préoccupations nécessaires à l'application. Le formalisme UML, comme le paradigme objet, ne permettent pas de séparer efficacement les préoccupations toujours à cause des problèmes de couplage.

Notre modèle de séparation et de composition des préoccupations peut donc être utilisé dans le cadre d'un développement dirigé par les modèles, moyennant des extensions. Cette utilisation

pourra intervenir en particulier à deux niveaux : i) au niveau du PIM, modèle représentant l'application pour séparer les différentes préoccupation, et ii) au niveau de la transformation PIM vers PSM pour faciliter la transformation et pouvoir décrire cette dernière de manière programmative.

Chapitre 9

Conclusion

Nous nous sommes intéressés dans cette thèse à un problème : la réutilisation des préoccupations dans les langages fortement typés à classes. Nous avons montré que la réutilisation d'une préoccupation est facilitée si cette dernière possède son protocole de composition. Ce protocole est encapsulé dans une entité. Cette dernière peut être ensuite spécialisée par héritage pour réutiliser la préoccupation dans un contexte donnée. De plus, la réutilisation d'une préoccupation peut aussi être facilitée par les adaptations disponibles ; ce sont les opérateurs d'adaptation qui modifient la préoccupation pour qu'elle se compose convenablement avec le reste de l'application.

Nous avons construit un nouveau modèle de séparation des préoccupations pour les langages à objet car aucun des modèles existants ne proposait toutes les propriétés que nous avions identifiées auparavant. Ce modèle à été présenté de manière détaillée indépendamment du langage utilisé. Il a été implémenté sous la forme d'un plug-in pour Eclipse (nommé JAdapt) car nous pensons que son utilisation à l'intérieur d'un environnement de développement simplifie la tâche de l'utilisateur.

La validation de JAdapt repose sur les mêmes exemples qui ont été utilisés pour montrer les avantages et les inconvénients des modèles existants. JAdapt encapsule correctement les protocoles de composition des préoccupations utilisées en exemple tout au long de cette thèse. De plus, les préoccupations sont composées convenablement grâce à un ensemble varié d'opérateurs d'adaptation.

En plus des résultats obtenus, une autre contribution de cette thèse réside dans les méthodes employées. Nous avons montré qu'il était possible, par des exemples simples, de mettre en évidence les problèmes de réutilisation des préoccupations. Nous avons également montré comment utiliser une plate-forme de développement pour enrichir un langage de programmation ; c'est une autre manière d'implémenter des langages de programmation sans recourir aux outils classiques pour construire des compilateurs.

Finalement, nous avons proposé des améliorations envisageables pour notre implémentation et son support dans l'environnement de développement Eclipse. Nous avons donné quelques exemples de réutilisation de notre modèle dans le paradigme des composants et dans l'approche MDA.

Chapitre 10

Bibliographie

1 [AB 91] Abiteboul S., Bonner A.J., « Objects and Views », Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, 1991.

2 [ABV 92] Aksit M., Bergmans L., Vural S., « An Object-Oriented Language-Database Integration Model : The Composition-Filters Approach », ECOOP'92, 1992.

3 [ACD 01] Attali I., Courbis C., Degenne P., Fau A., Parigot D., Pasquier C., « SmartTools: a Generator of Interactive Environments Tools », International Conference on Compiler Construction CC'01, ETAPS'2001, 2001.

4 [ANT 04] ANTLR, http://www.antlr.org.

5 [ASP 03] Aspect J, http://www.aspectj.org/.

6 [BA 01] Bergmans L., Aksit M., « Composing Multiple Concerns Using Composition Filters », Communications of the ACM, October 2001.

7 [BA 99] Bergmans L., Aksit M., « Analyzing Multi-Dimensional Programming in AOP and Composition Filters », Workshop on Multidimensional Separation of Concerns, OOPSLA'99, 1999.

8 [BC 90] Bracha G., Cook W., « Mixin-Based Inheritance », ECOOP/OOPSLA 90, 1990.

9 [BC 96] Bardou D., Cony C., « Split Objects: A Disciplined Use of Delegation within Objects », OOPSLA 96, 1996.

10 [BCS 02] Bruneton R., Coupaye T., Stefani J.B., « Recursive and Dynamic Software Composition with Sharing », Workshop WCOP, ECCOP 02, 2002.

11 [BDM 73] Birtwistle G.M., Dahl O.J. Myhrhaug B., Nygaard K., *SIMULA begin*, Philadelphia, Auerbach, 1973.

12 [Ber 98] Berger L., « Compile time and runtime reflection for dynamic evaluation of messages : Application to interactions between remote objects », Workshop on Reflective Programming in C++ and Java, OOPSLA'98, 1998.

13 [Ber 94] Bergmans L., The Composition-Filters Object Model, These de doctorat, Université de Twente, 1994.

14 [Beu 99] Beugnard A., « How to Make Aspects Reusable; A Proposition », Workshop on Aspect-Oriented Programming, ECOOP 99, 1999.

15 [BFJ 98] Brant J., Foote B., Johnson R.E., Roberts D., « Wrappers to the Rescue ». ECOOP 98, 1998.

16 [BL 01] Bouraqadi-Saâdani N.M.N., Ledoux T., « Le point sur la programmation par aspects », Technique et science informatiques, volume 20 – n° 4/2001, pages 505 à 528.

17 [BLR 98] Bouraquadi-Saâdani M.N., Ledoux T., Rivard F., « Safe Metaclass Programming », OOPSLA 98, 1998.

18 [BMV 00] Brichau J., De Meuter W., De Volder K., « Jumping aspects ». Workshop on Aspects and Dimensions of Concerns, ECOOP 2000, Cannes, France, Juin 2000.

19 [Bos 97] Bosch J, « Adaptating Object-Oriented Components », Workshop WCOP ECOOP 97, 1997.

20 [Bou 00] Bouraqadi N., « Concern Oriented Programming using Reflection », Workshop on Advenced Separation of Concerns in Object-Oriented Systems, OOPSLA'00, 2000.

21 [BRL 98] Bouraquadi-Saâdani M.N., Rivard F., Ledoux T., « Composition de Métaclasses », Journées Francophones des Langages Applicatifs, JFLA98, 1998.

22 [Bus 00] Bussard L., « Towards a pragmatic composition model of Corba services based on AspectJ ». Workshop on the Aspects & Dimensions of Concerns, ECOOP 00, 2000.

23 [Car 01] Caro P.S., « Adding Systemic Crosscuting and Super-Imposition to Composition Filters », Thèse de doctorat, Université de Twente, 2001.

24 [CE 99] Czarneci K., Eisenecker U., « Synthesizing Objects », ECOOP 1999, 18-42.

25 [CG 99] Carver L., Griswold W.G., « Sorting out Concerns », Workshop on Multi-Dimensional Separation of Concerns, OOPSLA'99, 1999

26 [Chi 00] Chiba S., « Load-time Structural Reflexion in Java », ECOOP'00, 2000.

27 [CLB 01] Coupaye T., Lenglet R., Beauvois M., Bruneton E., Déchamboux P., « Composant et composition dans l'architecture des systèmes répartis », Journée des Composants 2001, 2001.

28 [Coi 87] Cointe P., « The ObjVlisp Kernel : a Reflexive Lisp Architecture to define a Uniform Object-Oriented System », In P. Maes and D. Nardi, editors, Meta-Level Architectures and Reflection, North-Holland, Amsterdam, 1987.

29 [CSE 01] Constantinides C. A., Skotiniotis T., Elrad T., « Providing Dynamic Adaptability in an Aspect-Oriented Framework », Workshop Advanced Separation of Concern, ECOOP 2001, 2001.

30 [DEV 01] De Volder D., « Code Reuse, an Essantial Concern in the Design of Aspect Languages ? », Workshop Advanced separation of Concern, ECOOP 01, 2001.

31 [Dij 76] Dijkstra E.W., *A Discipline of Programming*, Prentice-Hall, Englewood cliffs, N.J., 1976

32 [DL 02] David P.C., Ledoux T., « Dynamic Adaptation of Non-Fonctionnal Concerns », Workshop unanticipated Software Evolution, ECOOP 02, 2002.

33 [DLB 01] David P.C., Ledoux T., Bouraqadi-Saâdani N.M.N. « Two-Step Weaving with Reflexion using AspectJ », Workshop Advanced Separation of Concern, OOPSLA 01, 2001.

34 [DMS 01] Douence R., Motelet O, Südholt M, « Sophisticated crosscuts for e-commerce », Workshop Advanced Separation of Concern, ECOOP 01, 2001.

35 [Duc 99] Ducasse S., « Evaluating Message Passing Control Techniques in Smalltalk », Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, SIGS Press, Juin 1999.

36 [ECL 04] Eclipse, http://www.eclipse.org.

37 [EL 03] Ernst E., Lorenz D.H., « Aspects and Polymorphism in AspectJ », AOSD 03, 2003.

38 [FS 98] Fradet P., Südholt M, « AOP: towards a generic framework using program transformation and analysis », Workshop Aspect Oriented Programming, ECOOP 98, 1998.

39 [FSJ 99] Fayad M. E., Schmidt D. C., Johnson R. E., *Building Application Frameworks*, Addison-Wesley Publishing Co., 1999.

40 [GHJ 99] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Catalogue de modèles de conception réutilisables*, Addison-Wesley Publishing Co., 1999.

41 [Gla 95] Glandrup M.H.J., Extending C++ using the concepts of composition filters, These de doctorat, Université de Twente, 1995.

42 [GR 83] Goldberg A., Robson D., *Smalltalk-80 : The Language and its Implementation*. Computer Science. Addison-Wesley Publishing Co., 1983.

43 [Gra 89] Graube N., « Metaclass Compatibility », OOPSLA'89, 1989.

44 [GS 96] Gottlob G., Schrefl M., « Extending Object-Oriented Systems with Roles », ACM Transactions on Information Systems (TOIS), 1996.

45 [Gyb 02] Gybels K., « Using a logic language to express cross-cutting through dynamic join points », Second German Workshop On Aspect-oriented Programming, ECOOP 1998, 1998.

46 [Har 01] Harrison W., « Composition and Multiple-Inheritance in OO Design (Where in the Madness is the Method?) », Workshop Advanced Separation of Concern, OOPSLA 01, 2001.

47 [Hau 01] Haupt M., « Concern Integration with JADE », PhDOOS ECOOP 2001, 2001.

48 [HB 02] Hachani O., Bardou D., « Using Aspect-Oriented Programming for Design Patterns Implementation », Workshop Reuse, OOIS 02, 2002.

49 [HK 02] Hannemann J., Kiczales G., « Design Pattern Implementation in Java and AspectJ », OOSPLA 02,2002.

50 [HLS 97] De Hondt K, Lucas C, Steyeart P., « Reuse Contracts as Component Interface Descriptions », Workshop WCOP, ECOOP 97, 1997.

51 [HM 01] Herrmann S., Mezini M., « Combining Composition Styles in the Evolvable Language LAC », Workshop Advanced Separation of Concern, ICSE 2001, 2001.

52 [HO 93] Harrison W., Ossher H., « Subject-Oriented Programming (A Critique of Pure Objects) », OOPSLA 93, 1993.

53 [HU 01] Hanenberg S., Unland R., « Using and Reusing Aspects in AspectJ », Workshop Advanced separation of Cconcern, OOSPLA 01, 2001.

54 [HYP 03] The HyperSpace HomePage, http://www.research.ibm.com/hyperspace/.

55 [JAC 03] Java Aspect Component, http://jac.aopsys.com.

56 [JAS 04] JAsCo, http://jssel.vub.ac.be/jasco/.

57 [JAV 04] Java, http://java.sun.com.

58 [JAX 04] Java Architecture for XML Binding (JAXB), http://java.sun.com/xml/jaxb/index.jsp.

59 [JCC 04] Javacc, http://javacc.dev.java.net.

60 [Kee 89] Keene S. E., *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley (Reading, Massachusetts, 1989).

61 [Ken 00] Kendall E. A., « Reengineering for Separation of Concerns », OOSPLA'00, 2000.

62 [KLM 97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.V., Loingtier J.M., Irwin J., « Aspect Oriented Programming », ECOOP 97,1997.

63 [KN 74] Kathleen J., Niklaus W., *Pascal user manual and report*, Lecture notes in computer science - LNCS, 0302-9743, Springer, 1974.

64 [KRB 91] Kiczales G., Des Rivières J., et Bobrow D.G., *The Art of the Métaobjet Protocol*, MIT press, Cambridge, Massachusetts, 1991. ISBN 0-262-11158-6.

65 [LH 95] Lopes C.V., Hursch W., Separation of Concerns, Technical Repport NU-CCS-95-03, Northeastern University, Boston, February 1995. (disponible à http://citeseer.nj.nec.com/lopes95separation.html).

66 [LLM 99] Lieberherr K., Lorenz D., Mezini M., Programming with aspectual components, Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, Avril 1999.

67 [MDA 04] Model Driven Architecture, http://www.omg.org/MDA.

68 [MH 03] McDirmid S., Hsieh W.C., « Aspect-Oriented Programming with Jiazzi », Aspect Oriented Software Development, AOSD 2003, 2003.

69 [MMC 95] Mulet P., Malenfant J., Cointe P., « Towards a Methodology for Explicit Composition of MetaObjects », OOPSLA 95, 1995.

70 [MO 02] Mezini M., Ostermann K., « Integrating Independent Components with On-Demand Remodularization », OOPSLA 2002, 2002.

71 [MO 03] Mezini M., Ostermann K., « Conquering Aspects with Casear », Aspect Oriented Development AOSD'03, 2003.

72 [MS 97] Mikhajlov L., Sekerinski E., « The fragile base class problem and its impact on Component Systems », Workshop WCOP, ECOOP 97, 1997.

73 [MS 98] Mikhajlov L., Sekerinski E., « A Study of The Fragile Base Class Problem », ECCOP 98, 1998.

74 [MSC 93] ITU-TS, *Recommandation Z.120 : Message Senquence Charts (MSC).*, Genève, Septembre 1993.

75 [MSL 00] Mezini M., Seiter L., Lieberherr K., *Component Integration with Pluggable Composite Adapters*, Software Architectures and Component Technology: The State of the Art in Research and Practice, Kluwer Publishing, 2000.

76 [Nak 00] Nakajima S. « Separation Of Concerns in early stage of Framework development », ECOOP 2000.

77 [NK 01] Noda N., Kishi T., « Implementing design patterns using advanced separation of concerns », Workshop Advanced separation of Concern, OOSPLA 01, 2001.

78 [OHBS 94] Ossher H., Harrison W., Budinsky F. , Simmonds I., « Subject-Oriented Programming : Supporting Decentralized Development of Objects », Proceedings of the 7th IBM Conference on Object-Oriented Technology, Juillet 1994.

79 [OK 00] Ostermann K., Kniesel G., « Independent Extensibility – an open challenge for AspectJ and Hyper/J », Workshop Aspect and Dimension of Concern, ECOOP 00, 2000.

80 [OKHKK 95] Ossher H., Kaplan M., Harrison W., Katz A., Kruskal V., « Subject-Oriented Composition Rules », OOPSLA 95, 1995.

81 [OM 01] Ostermann K., Mezini M., « Object-Oriented Composition is Tangled », Workshop Advanced Separation of Concern, ECOOP 01, 2001.

82 [Ost 02] Ostermann K., « Dynamically Composable Collaborations with Delagation Layers », ECOOP 02, 2002.

83 [OT 00a] Ossher H., Tarr P., « Hyper/J: Multi-Dimentionnal Separation of Concern for Java », *ICSE 00*, 2000.

84 [OT 00] Ossher H., Tarr P., « Multi-Dimensional Separation of Concerns and The Hyperspace Approach », Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

85 [Par 72] Parnas D.L., On the criteria to be used in decomposing systems into modules, Communications of the ACM, 15(12):1053-1058, December 1972.

86 [Paw 02] Pawlak R., La Programmation par Aspects Interactionnelle pour la construction d'Application à Préoccupations Multiples, Thèse de Doctorat, CNAM-CEDRIC, Decembre 2002.

87 [PSD 01] Pawlak R., Seinturier L., Duchien L., Florin G., « JAC : A Flexible Efficient Solution for Aspect-Oriented Programming in Java », Reflexion'01, 2001.

88 [PSD 01a] Pawlak R., Seinturier L., Duchien L., Florin G., « Dynamic Wrappers : Handling he Composition Issue with JAC », TOOLS'01, 2001.

89 [RC 03] Rashid A., Chitchyan R., « Persistance as an Aspect », AOSD 03, 2003.

90 [SB 98] Smaragdakis Y., Batory D., « Implementing Layered Designs with mixin Layers », ECOOP 98, 1998.

91 [SCH 04] Schema XML, http://www.w3.org/XML/Schema.

92 [SIN 04] The Sina Language, http://trese.cs.utwente.nl/sina/.

93 [SMM 03] Ségura-Devillechaise M., Menaud J.M., Muller G., Lawall J.L., « Web Cache Prefetching as an Aspect : Towards a Dynamic-Weaving Based Solution», AOSD 03, 2003.

94 [SVJ 03] Suvée D., Vanderperren W., Jonckers V., « JasCo: an Aspect-Oriented approach tailored for Component Based Software Development», Aspect Oriented Software Development 03, 2003.

95 [Szy 98] Szyperski C., *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley Publishing Co, (Readgind, MA), 1998..

96 [THO 00] Tarr P., Harrison W., Ossher H., Finkelstein A., Nuseibeh B., Perry D., Summary of Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, ISCE 2000.

97 [TSC 01] Tatsubori M., Sasaki T., Chiba S., Itano K., « A Bytecode Translator for Ditribued Execution of " Legacy " Java Software », ECOOP'01, 2001.

98 [UML 04] Unified Modeling Langage, http://www.omg.org/UML.

99 [VD 98] De Volder K., D'Hondt T., « Aspect-Oriented Logic Meta Programming », Workshop Aspect-oriented Programming, ECOOP 1998, 1998.

100 [VW 02] Vanderperren W., Wydaeghe B., « Separating concerns in a high-level component-based context », Workshop EasyComp, ETAPS 2002, 2002.

101 [VWD 01] Vanhaute B., De Win B., De Decker B., « Building Frameworks in AspectJ », Workshop Advanced Separation of Concern, ECOOP 01, 2001.

102 [Wic 99] Whishman J.C., « ComposeJ The development of a preprocessor to facilitate Composition Filters in the Java Language », Thèse de Doctorat, Université de Twente, 1999.

103 [XML 04] XML, http://www.w3.org/XML/.

Résumé

Cette thèse a pour sujet la réutilisation des préoccupations dans le paradigme de l'objet. De nombreux langages et modèles pour la séparation des préoccupations existent dans la littérature. Leurs limitations nous ont permis d'isoler toutes les propriétés nécessaires pour réutiliser facilement les préoccupations. Ces propriétés sont principalement : *i)* l'encapsulation de la composition au sein d'entités réifiées (qui doivent pouvoir être abstraite, et supporter l'héritage) et *ii)* un panel assez varié d'opérateurs d'adaptation (encapsulé par les entités précédemment citées) nécessaire à la composition de préoccupations dans l'objet. Aucun langage ou modèle de la littérature ne procure actuellement ces propriétés. Nous avons donc construit un modèle réunissant toutes ces propriétés, qui a été implémenté sous la forme d'un plug-in pour l'environnement de développement Eclipse. Nous avons validé notre implémentation sur les mêmes exemples qui ont été utilisés pour dresser les manques de la littérature. Enfin, nous évoquons les perspectives d'utilisation de notre modèle dans le paradigme des composants et dans l'approche MDA.

Mots-clés — Séparation des préoccupations, programmation orientée objets, réutilisation de code, environnements compilés et typés.

Abstract

This thesis deals with reuse of concerns in the object-oriented paradigm. Many languages and models for separation of concerns exist in the literature. Their weakness permit us to isolate all the required properties in order to easily separate and reuse concerns. The most important properties are: *i)* encapsulation of the composition in reified entities (which must support abstractness and inheritance) and *ii)* a lot of adaptation operators (encapsulated by the reified entities previously quoted) needed in order to object-oriented composition of concerns. No language or model from literature have these properties. Therefore, we propose a model featuring all of these properties. This model has been implemented as a plug-in for Eclipse. We validated our implementation with the same examples previously used to reveal the miss of literature. Finally, we identified some future uses of our model in component-oriented programming and model driven architecture.

Keywords — Separation of concerns, object-oriented programming, code reuse, compiled and typed environments.